Министерство образования и науки Российской Федерации
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Т. В. Коликова

ОПЕРАЦИОННЫЕ СИСТЕМЫ.

УЧЕБНОЕ ПОСОБИЕ

Санкт-Петербург
2012

СОДЕРЖАНИЕ

Предисловие

Как изучать операционные системы (ОС)? Сначала нужно разобраться, зачем нужны ОС, как они устроены изнутри, как их проектировать и разрабатывать.

Одних теоретических знаний недостаточно. Можно считать, что вы по настоящему разобрались с ОС, только если вам удается применить полученные теоретические знания на практике. Разработчики программного обеспечения (ПО) должны уметь создавать эффективные приложения с использованием сервисов, предоставленных непосредственно ОС.

Мы будем изучать, как ОС Windows и UNIX спроектированы и реализованы, а на лабораторных работах писать эффективные программы, которые напрямую, используют возможности этих ОС. Для Windows будем использовать Win32 API, а для UNIX системные вызовы.

Лекции содержат в основном теоретический материал, а в описаниях к лабораторным работам приводятся необходимые Win32 API функции и системные вызовов Unix. Задания приводятся 3-х уровней сложности. Вся информация, которая нужна для выполнения заданий первого уровня, есть в лекциях и описаниях лабораторных работ. Кроме того, для многих заданий этого уровня приводятся "скелеты кода". Следующие уровни сложности потребуют от вас самостоятельных решений и изучения MSDN.

Написано много книг посвященных ОС, но они дорогие и толстые, кроме того, одной книги будет недостаточно. Рассмотрим основные типы существующих изданий, их достоинства и недостатки:

- Теоретические основы построения ОС. Они не позволяют научиться на практике, применять полученные знания.
- Основы системного администрирования. В этих книгах обычно описывается только процесс администрирования и подразумевается, что необходимыми теоретическими знаниями читатель уже обладает.
- Разработка эффективных приложений. Теоретические основы построения ОС изложены очень коротко.

Данный учебник - это попытка совместить теорию и практику и осуществить быстрый старт. А дальше вы должны идти самостоятельно, уже осмысленно используя MSDN. Очень хочется, чтобы вам было интересно, и энтузиазм вас не покидал.

Основные понятия и определения

Любая вычислительная система состоит из ***аппаратного обеспечения*** (***hardware***) и ***программного обеспечения*** (***software***). Программное обеспечение (ПО), в свою очередь, делят на *прикладное* и *системное*. Прикладное ПО - это прикладные программы. Системное ПО - это программы, способствующие функционированию и разработке прикладных программ. ***Операционная система*** (***ОС***) - это основной компонент системного ПО.

Операционная система - это программа, обеспечивающая среду выполнения для других программ и облегчающая им доступ к устройствам (процессоры, диски и т.д.). Она очень удобна, но не абсолютно необходима. На заре компьютерной эпохи загружали программу в память с перфоленты, задавали вручную стартовый адрес, и с него начиналось выполнение.

Современные ОС предоставляют пользователям два вида услуг. Во-первых, они упрощают использование аппаратных средств. Создаваемая ими виртуальная машина заметно отличается от реальной. ОС изолируют пользователей от непонятной им части аппаратной части компьютеров.

Во-вторых, ОС обеспечивает распределение вычислительных ресурсов между пользователями. Один из самых важных ресурсов - процессор (процессорное время).

В ***многозадачной*** (***multitasking***) ОС выполняемая работа подразделяется на ***процессы*** (***processes***), каждому из которых предоставляется память, системные ресурсы и, по крайней мере, один ***поток управлени***я (***thread of execution***).

Процесс (***process***) - логическая единица работы в ОС. Процесс включает в себя виртуальное адресное пространство, исполняемую программу, один или несколько потоков управления, а также системные ресурсы, которые ОС выделяет потокам процесса ОС.

Поток исполнения (***thread of execution***) - исполняемая сущность внутри процесса. Поток состоит из указателя текущей команды, пользовательского стека, стека ядра и набора значений регистров. Все потоки процесса имеют доступ к его адресному пространству и другим ресурсам.

ОС выполняет один поток в течение кванта времени, потом переключается на другой. Многозадачность очень полезна даже в однопользовательской системе, так как несколько задач будут выполняться одновременно.

Кроме того, ОС распределяет память и управляет доступом к файлам и устройствам. ОС различаются по способам, которыми они представляют виртуальную машину пользователям и распределяют между ними ресурсы. Мы будем рассматривать, как все это происходит в Windows и UNIX.

Архитектура ОС

Существуют различные способы структурирования кода ОС. Рассмотрим наиболее распространенные из них.

Монолитная, послойная и клиент-серверная модели

Один подход состоит в организации системы как набора процедур, каждую из которых может вызывать любая пользовательская программа. Это так называемая *монолитная структура*.

При таком подходе в различных участках кода используется информация об устройстве всей системы. Монолитную ОС трудно расширять, так как изменения в одной процедуре, могут вызвать ошибки в других частях системы. Во всех монолитных ОС приложения отделены от собственно ОС. Код ОС выполняется в ***привилегированном режиме*** (***режиме ядра***) и имеет доступ к данным системы и аппаратуре. Приложения выполняются в ***пользовательском режиме*** (***user mode***), в котором им предоставлен ограниченный набор интерфейсов и ограниченный доступ к системным данным.

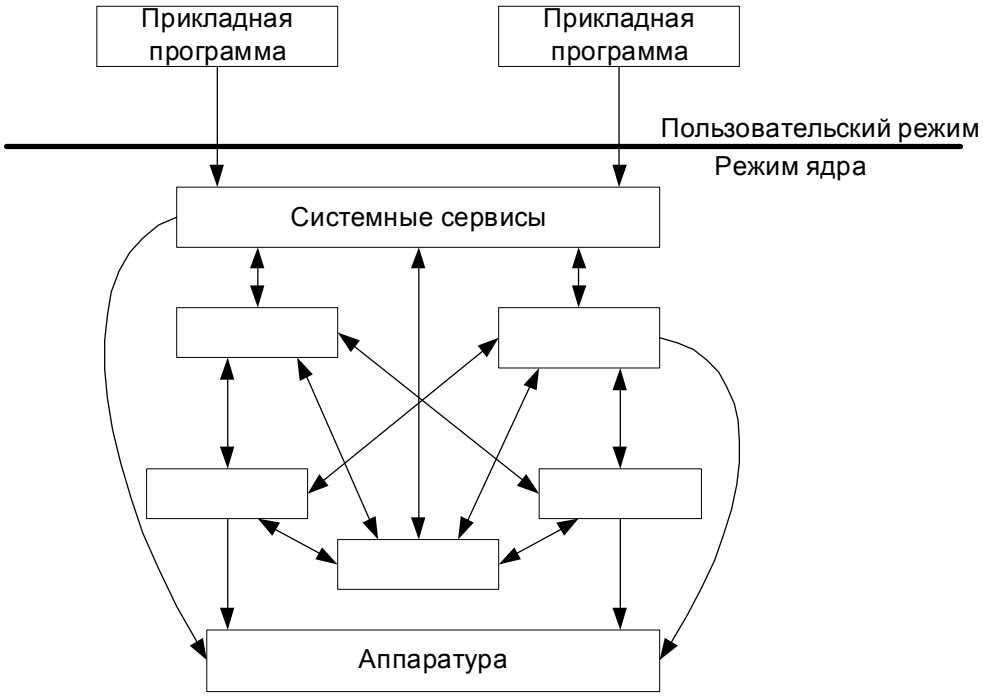
Пользовательский режим (***user mode***) - непривилегированный режим работы процессора, в котором исполняется код прикладных программ. Поток, исполняющийся в пользовательском режиме, может получить доступ к системным ресурсам только посредством вызова системных сервисов.

Когда программа пользовательского режима вызывает системный сервис, вызов перехватывается и вызывающий поток переключается в ***режим ядра*** (***kernel mode***).

Режим ядра (***kernel mode***) - привилегированный режим работы процессора, в котором исполняется код ОС. Поток, исполняющийся в режиме ядра, имеет доступ к системной памяти и аппаратуре.

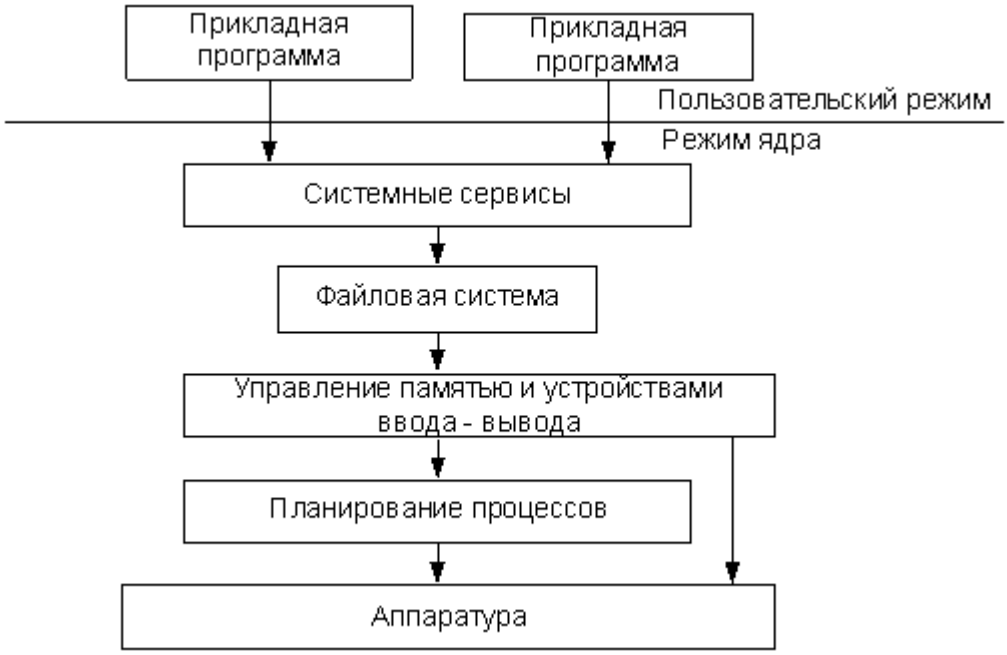
Когда выполнение системного сервиса завершается, ОС переключает поток обратно в пользовательский режим. Структура монолитной ОС показана на Рис. 1 (*Рисунок П.1*).

Рисунок П.1 Монолитная ОС



Другой подход к структурированию ОС предполагает разделение ее на слои (модули), расположенные один поверх другого. Каждый слой предоставляет набор функций, которые могут вызываться вышестоящим слоем. А код расположенный в некотором слое может вызывать функции только нижележащего слоя. Таким образом, преимущество послойной организации ОС в том, что код каждого слоя получает доступ только к необходимым ему интерфейсам нижележащих слоев. Каждый слой не получает неограниченную власть. Такая структура облегчает отладку и расширение ОС. Можно отладить или заменить любой слой, не затрагивая остальных частей. Одна из возможных послойных структур показана на рисунке (*Рисунок П.2*).

Рисунок П.2 Послойная ОС

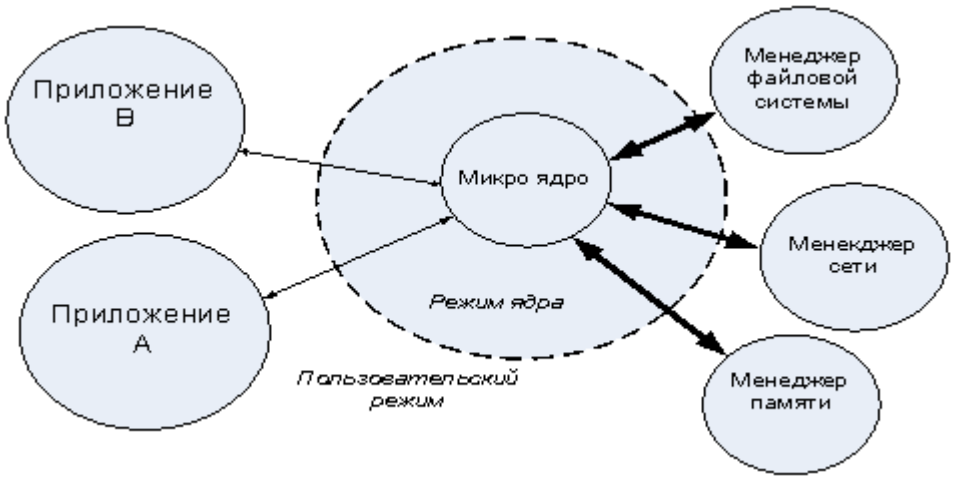


Третий подход к структурированию ОС - это модель клиент-сервер. В этом случае ОС состоит из нескольких процессов (серверов), каждый из которых реализует определенный набор сервисов: сервер памяти, сервер процессов и т.д.

Каждый сервер выполняется в пользовательском режиме и занимается обслуживанием клиентов. Клиентом может быть другой компонент ОС или прикладная программа. Клиент запрашивает выполнение сервиса, посылая серверу сообщение. Ядро (или микро ядро) ОС, выполняющееся в режиме ядра, доставляет сообщение серверу, тот выполняет запрашиваемое действие, после чего ядро возвращает клиенту результат в виде другого сообщения. Структура такой системы показана на рисунке (*Рисунок П.3*).

В этом случае ОС состоит из автономных компонентов небольшого размера. Так как все серверы выполняются как отдельные процессы пользовательского режима, то нарушение работы одного из них не нарушит работу остальных частей ОС. Кроме того, разные серверы могут выполняться на разных процессорах одного компьютера или на разных компьютерах, а это делает ОС пригодной для распределенных вычислений.

Рисунок П.3 Клиент-серверная ОС



Модель, показанная на рисунке (*Рисунок П.3)*, - это идеализированная система клиент-сервер, так как ядро состоит только из средств передачи сообщений. В действительности существуют как системы, которые выполняют в режиме ядра малый объем работы, так и те, которые выполняют большой объем работы. Например, в ОС Mach используется ядро минимального размера. Оно выполняет планирование потоков, передачу сообщений, управление виртуальной памятью, а также включает драйверы устройств. Интерфейсы прикладных программ (API), файловые системы и сетевая поддержка работают в пользовательском режиме.

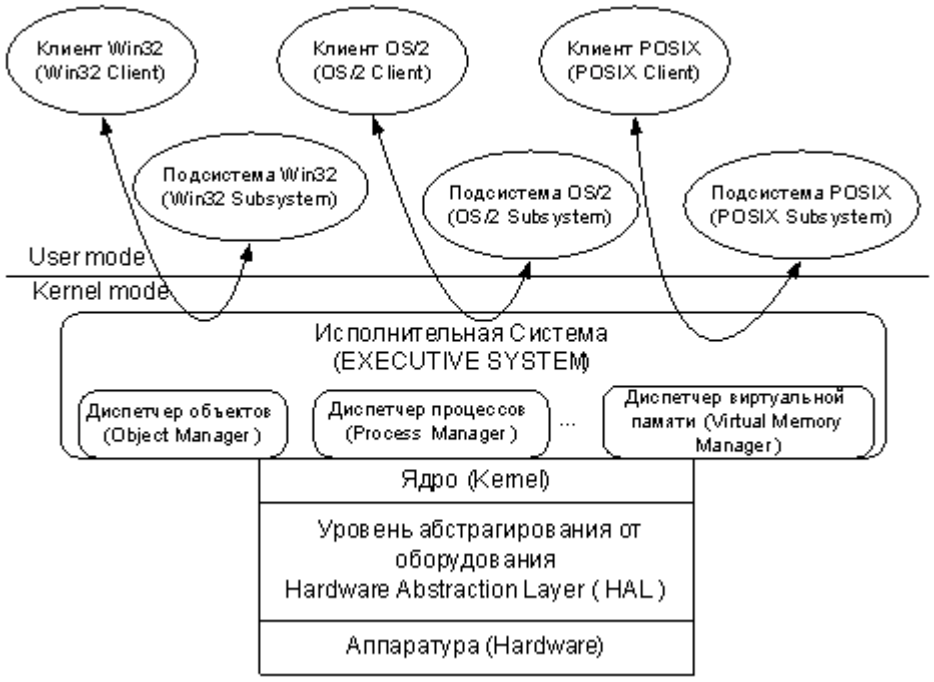
В структуре Windows NT используются элементы как послойной, так и клиент серверной архитектуры (*Рисунок П.4*).

Объектная модель

В ОС, как и в случае других больших программ, трудно найти одну главную программу, которая управляет всей ОС. Если это можно сделать, то систему разрабатывают сверху вниз. При использовании объектно-ориентированной методологии, сначала рассматривают данные, с которыми должна работать программа для выполнения своей задачи. Для ОС такими данными являются системные ресурсы - файлы, процессы, блоки памяти и т.д.

Основная цель любой разработки - это создание программного обеспечения, которое легко изменять. Это особенно важно, так как по статистике 70% цены программных продуктов приходится на сопровождение. Сопровождение включает добавление новых возможностей, модификацию форматов данных и исправление ошибок.

Рисунок П.4 Послойная и клиент-серверная структура Windows



Использование объектно-ориентированного подхода, позволяет скрыть физическое представление данных внутри объектов. ***Объект*** (***object***) - это структура данных, физический формат которой скрыт в определении типа. Он имеет набор *атрибутов* (*attributes*) и с ним работает группа *сервисов* (*services*).

ОС использует объекты для представления системных ресурсов. Каждый системный ресурс, который могут совместно использовать несколько процессов, реализован как объект и обрабатывается объектными сервисами. Например, если произошли изменения в аппаратуре, то необходимо изменить только объект, представляющий данный аппаратный ресурс, и его сервисы. Код, который использует объект, модифицировать не нужно.

Кроме того, использование объектов дает ряд дополнительных преимуществ:

- Доступ ОС к ресурсам и работа с ними унифицирована. Каждый ресурс - это объект, поэтому контроль использования ресурсов сводится к отслеживанию создания и использования объектов. Создание, удаление и ссылка на любой объект осуществляется одинаково с помощью *описателей* (*handles*) объектов.

- Упрощается защита - для всех объектов она осуществляется одинаково. При попытке доступа к объекту подсистема защиты проверяет допустимость операции независимо от типа объекта.

- Объекты предоставляют унифицированную парадигму для совместного использования ресурсов несколькими процессами. Для работы с объектами любого типа используются описатели объектов. Два процесса совместно используют объект тогда, когда каждый из них открыл его описатель. ОС отслеживает число открытых описателей для данного объекта, чтобы определить, действительно ли он используется или его можно удалить.

Классификация ОС

По числу одновременно работающих пользователей ОС можно разделить на два класса:

- Однопользовательские (MS-DOS, Windows 3.x);
- Многопользовательские (Windows NT, UNIX).

По числу одновременно выполняемых задач ОС можно разделить на два класса:

- Многозадачные (Unix, OS/2, Windows);
- Однозадачные (например, MS-DOS).

Многозадачность - это техника, применяемая ОС для использования одного процессора нескольким потоками управления. Если у компьютера имеется более одного процессора, то необходимо перейти к модели *мультипроцессорной обработки (multiprocessing)*. Многозадачная ОС создает иллюзию одновременного выполнения нескольких потоков. А в ОС с мультипроцессорной обработкой в действительности выполняются несколько потоков одновременно (по одному потоку на процессор).

ОС с мультипроцессорной обработкой делятся на две категории:

- *С асимметричной мультипроцессорной обработкой (asymmetric multiprocessing, ASMP).* Код ОС выполняется на одном процессоре, а на других процессорах выполняются пользовательские задачи (*Рисунок П.5*).
- *С симметричной мультипроцессорной обработкой (symmetric multiprocessing, SMP).* Код ОС будет выполняться на любом свободном процессоре или на всех процессорах одновременно (*Рисунок П.6*).

Рисунок П.5 Асимметричная мультипроцессорная обработка

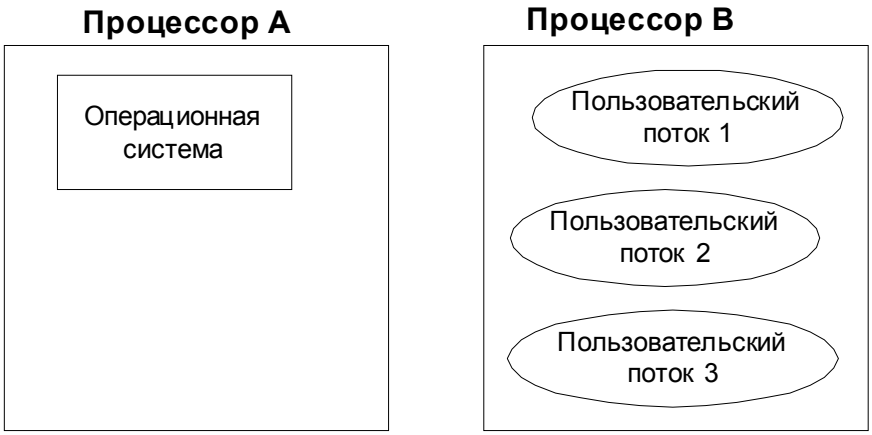
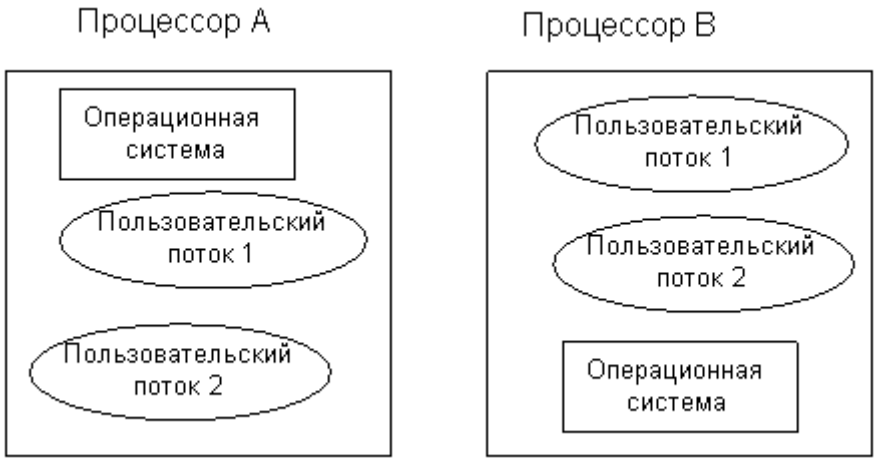


Рисунок П.6 Симметричная мультипроцессорная обработка



Введение

В мире ОС прогресс достигается медленно. Разработка ОС занимает несколько лет. Созданная ОС, бесполезна, пока не появятся приложения, позволяющие эффективно использовать ее возможности. Кроме того, нужно научиться использовать саму ОС. Получается, что мы работаем с "пожилыми" ОС (как минимум 10-ти летней давности).

Аппаратные технологии развиваются намного быстрее. Появляются компьютеры с более мощными процессорами, большим объемом памяти, с несколькими процессорами и т.д. А разработчики пытаются усовершенствовать старые ОС, чтобы те могли использовать новые возможности аппаратных средств. Гораздо лучше создать ОС, которая будет поддерживать новые аппаратные разработки.

Требования рынка

Сформулируем ***основные требования*** (***requirements***), предъявляемые рынком к ОС:

- ***Переносимость***. Новшества в аппаратном обеспечении возникают быстро и часто непредсказуемо. Написание ОС на переносимом языке позволило бы быстро переходить от одной архитектуры к другой.
- ***Мультипроцессорная обработка и масштабируемость***. Компьютеры с несколькими процессорами регулярно появляются на рынке, но лишь немногие из существующих ОС могут в полной мере использовать их возможности. Создание ОС как масштабируемой многопроцессорной ОС позволило бы запускать одно и то же приложение, как на однопроцессорных, так и на многопроцессорных машинах. Несколько приложений выполнялись бы одновременно с максимальной скоростью, а приложения, требующие большого объема вычислений, могли бы повысить свою производительность, распределив работу между несколькими процессорами.
- ***Распределенные вычисления***. В 80-е годы персональные компьютеры стали широко доступны. Там где раньше использовали большую ЭВМ и терминалы, стали использовать персональные компьютеры, объединенные в сеть. Это позволило им совместно использовать аппаратные и вычислительные ресурсы (файл-сервер, сервер печати, сервер вычислений). Функции поддержки сети необходимо встроить непосредственно в саму ОС.

- ***Совместимость с POSIX.*** Во второй половине 80-х годов POSIX был определен в качестве стандарта, которому должно удовлетворять программное обеспечение, поставляемое по правительственным контрактам в США. POSIX (portable operating system interface based on UNIX) - это набор международных стандартов для интерфейсов ОС UNIX-типа. Соблюдение стандартов позволяет легко переносить приложения с одной системы на другую. Чтобы удовлетворять требованиям к поставкам по заказам правительства, NT должна была обеспечить среду исполнения приложений POSIX.

- ***Защита от несанкционированного доступа.*** Кроме совместимости с POSIX, правительство США устанавливает правила защиты для приложений, используемых в правительственных учреждениях. Система должна пройти правительственную сертификацию. Правила защиты включают обязательные требования, такие как защита ресурсов пользователя от других пользователей и возможность установления квот на системные ресурсы для предотвращения захвата одним пользователем всех системных ресурсов. Система защиты ОС должна иметь уровень C2, определенный министерством обороны США как обеспечивающий селективное назначение прав доступа владельцем, включение возможностей аудита для учета субъектов и инициируемых ими действий. Владелец системного ресурса может определять, кто имеет доступ к ресурсу, а ОС в состоянии определить, когда и кем осуществлена попытка доступа к ресурсу.

Свойства ОС

Свойства, которыми должна обладать современная ОС:

- ***Расширяемость.*** Код должен быть написан так, чтобы его удобно было дополнять и модифицировать при изменении требований рынка.

- ***Переносимость.*** Код должен легко переноситься с одного процессора на другой.

- ***Надежность и безопасность.*** Система должна защищать себя как от внутренних сбоев, так и от внешнего вторжения. Она должна всегда вести себя предсказуемо, и у приложений не должно быть возможности повредить ОС или нарушить ее функционирование.

- ***Совместимость.*** Пользовательский интерфейс и API должны быть совместимы с существующими системами.

- ***Производительность.*** Система должна быть максимально быстрой и обеспечивать минимальное время отклика на каждой аппаратной платформе.

Далее рассмотрим, как разработчики Windows NT создали ОС, удовлетворяющую этим свойствам.

Расширяемость

Можно рассмотреть два аспекта расширяемости. Первый относится к конфигурации операционной системы. Машина с Windows можно сконфигурировать как рабочую станцию или как сервер. При любой конфигурации используется тот же основной исходный код, но во время компиляции включаются различные компоненты. Это позволяет оптимизировать систему при использовании на рабочей станции или на сервере, не создавая две различные ОС.

Второй более значительный аспект расширяемости - это послойно-модульная структура ОС (*Рисунок В.1*). Фундаментальные механизмы реализованы в ***ядре*** (***kernel***).

Это низкоуровневые, четко определенные и хорошо предсказуемые примитивы. Ядро реализует системные механизмы и не участвует в принятии решений, связанных с определением системной политики. Дополнительные механизмы затем реализуются на этой базе для задания определенной политики. Такой подход обеспечивает безопасность и надежность. ***Исполнительная система*** (***executive system***) спроектирована как уровень абстрагирования от ядра. Она обеспечивает специфические механизмы/политики для управления объектами и памятью, процессами, файлами и устройствами. Вместе ядро и исполнительная система реализуют основные функции ОС, которые снова расширяются посредством *подсистем* (*subsystems*).

Хотя ядро и исполнительная система спроектированы и реализованы как отдельные слои модули, они включаются в один выполняемый модуль **NTOSKRNL.EXE**. К нему также при необходимости подключаются ***динамически подключаемые библиотеки*** (***DLLs***).

Следующий уровень абстрагирования - это уровень подсистем, которые обеспечивают переносимость приложений. Подсистема - это программный модуль, который использует сервисы (механизмы), реализованные в ядре и исполнительной системе для реализации более абстрактных сервисов, в особенности сервисов, предоставляемых конкретной ОС. Например, NT 4.0 включает подсистему POSIX, которая выполняется на базе ядра и исполнительной системы и делает NT подобной POSIX. Подсистемы называют ***подсистемами окружения***

(***environment subsystems***) или ***защищенными подсистемами*** (***protected subsystems***). Другие подсистемы предоставляют специализированные сервисы, например подсистема безопасности ***Local Security Authority Server*** (**LSASS.EXE**).

Рисунок B.1 Структура Windows

Все подсистемы и все использующие их пользовательские приложения выполняются в пользовательском режиме. Подсистемы - это ключевой компонент, позволяющий поддерживать различные модели вычислений (например, MS-DOS или Win16). Приложения, написанные для MS-DOS, используют подсистему MS-DOS. Эта подсистема предоставляет такой же API, как и MS-DOS, давая возможность программам MS-DOS выполняться в NT.

Переносимость

Переносимость: система должна работать на разных аппаратных архитектурах и обладать способностью к достаточно легкому переносу на новые аппаратные архитектуры, если на рынке возникнет потребность в этом. Переносимость обеспечивает использование всей ОС целиком на машине с другим процессором или конфигурацией при минимальных изменениях исходного текста. Часто ОС делят на переносимые и непереносимые, хотя "перенести" можно любую программу, вопрос только в том насколько это сложно сделать.

Чтобы написать переносимую программу нужно следовать определенным правилам:

- Код должен быть написан на языке, доступном на всех машинах, на которые планируется переносить программу. Поэтому код следует писать на стандартизованном языке высокого уровня. Язык ассемблера по своей природе не переносим.

- Следует учесть, в какую среду планируется переносить программное обеспечение. Например, программу, написанную для 32-разрядной адресации, не удастся перенести на машину с 16-разрядными адресами - только ценой огромных усилий.

- Нужно минимизировать, а там где только возможно вообще удалить код, работающий непосредственно с аппаратурой (прямое манипулирование регистрами, другими аппаратными структурами, использование конкретной аппаратной конфигурации или емкости).

- Там где аппаратно-зависимого кода не избежать, его необходимо изолировать в небольшом количестве модулей.

Два последних правила тесно связаны друг с другом. Например, можно скрыть аппаратно-зависимую структуру в абстрактном типе данных. А другие модули будут работать с этим типом данных посредством процедур. Тогда при переносе программы необходимо изменить только этот тип данных и связанные с ним процедуры.

Чтобы упростить переносимость NT были использованы следующие подходы:

Переносимый язык С. Система написана в основном на языке С (стандарт ANSI). Этот язык стандартизован и для него широко доступны компиляторы. Небольшие части системы написаны на C++ (графические компоненты, фрагменты сетевого пользовательского интерфейса). Язык ассемблера использовали только для тех частей ОС, которые должны работать непосредственно с оборудованием, и для компонентов требующих максимальной скорости выполнения. Однако непереносимый код был тщательно изолирован внутри использующих его компонент.

Изоляция от процессора. Некоторые низкоуровневые фрагменты ОС должны работать с зависимыми от процессора структурами данных. Такой код поместили в небольшие модули, которые можно заменить аналогичными модулями для других процессоров.

Изоляция от платформы. Платформенно-зависимый код инкапсулирован внутри динамически подключаемой библиотеки, называемой *слоем абстрагирования от оборудования* (*hardware abstraction layer, HAL*). Платформенно-зависимыми называются

свойства, которые могут отличаться на компьютерах с одним и тем же типом процессора от разных производителей (кэш, контроллеры прерываний).

Переносимость во многих аспектах перекрывается с расширяемостью. Подсистемы обеспечивают расширяемость, позволяя выполнять различные приложения, но также они являются и основой для переносимости. Приложения, написанные для других ОС за счет подсистем можно легко перенести под NT. Кроме подсистемы для MS-DOS, подсистемы Win16 и POSIX основной подсистемой является подсистема Win32. В общем случае разработчики программного обеспечения могут разработать нужную подсистему, удовлетворяющую их требованиям к сервисам ОС, используя интерфейс исполнительной системы/ядра. Подсистема Win32 играет особую роль, так как реализует расширение исполнительной системы, необходимое другим подсистемам. Каждая из подсистем использует подсистему Win32, поэтому ее присутствие обязательно.

HAL, ядро и исполнительная система функционируют в режиме ядра и экспортируют API, который используют создатели подсистем. Создатели подсистем окружения выбирают целевой API (Win16, POSIX или OS/2 API) и строят подсистему, реализующий этот API, используя все сервисы, предоставляемые в режиме ядра. API Microsoft - это Win32 API, который предоставляется подсистемой Win32. Прикладные программисты используют интерфейс Win32 API, а не интерфейс **NTOSKRNL.EXE**.

Надежность и безопасность

Надежность: система должна быть защищенной как от внутренних сбоев, так и от внешних деструктивных действий. У приложений не должно быть возможности нарушить работу ОС или других приложений. Надежная ОС должна давать предсказуемый отклик на ошибочные состояния, даже если они вызваны сбоями аппаратуры. Кроме того, надежная ОС должна активно защищать себя и своих пользователей от случайного или умышленного вреда, наносимого пользовательскими программами.

Структурная обработка исключений (*structured exception handling*) - это метод перехвата ошибочных состояний и унифицированной их обработки. Он используется для защиты *Windows* NT от программных или аппаратных ошибок. Когда возникает ненормальное событие, ОС возбуждает исключение и автоматически вызывается код обработки исключения, гарантируя, что пользовательским программам и самой системе не будет нанесен вред.

Повышению устойчивости способствуют и другие свойства:

Внутренняя модель системы клиент/сервер. Ядро и исполнительная система работают в режиме ядра (*kernel mode*), а пользовательские приложения (и подсистемы) - в пользовательском режиме (*user mode)*. Приложения используют метод передачи сообщений для взаимодействия со службами нижнего уровня. Это существенно повышает надежность системы, но несколько снижает производительность из-за дополнительных затрат на передачу сообщений.

Многозадачность с вытеснением (**preemptive multi tasking**). Это гарантирует адекватное распределение процессора на протяжении работы системы, предотвращает монопольный захват процессора приложением и остановку системы, когда приложение работает нестабильно.

Модульная структура исполнительной системы. Отдельные модули взаимодействуют друг с другом только через тщательно разработанные программные интерфейсы. Какой-либо модуль можно заменить другим, реализующим те же самые интерфейсы.

Защита от несанкционированного доступа. Уровень защиты С2 правительства США, предоставляющий различные механизмы защиты: регистрация пользователей в системе, квоты на ресурсы и защита объектов.

Файловая система NT (NTFS). NTFS способна к восстановлению после всех типов дисковых ошибок, включая ошибки в критически важных секторах диска. Для обеспечения восстанавливаемости используется избыточное хранение данных и обработка транзакций. Транзакция - это некая совокупность операций, которая должна выполниться от начала до конца. Если одна или несколько операций из этой совокупности (транзакции) не выполнились, то система возвращается в исходную точку. Это позволяет отменить незавершенную или неправильную операцию, возникающую в случае аппаратного или программного сбоя.

Виртуальная память (**virtual memory**). Каждому приложению предоставляется личное адресное пространство. При обращении по виртуальным адресам диспетчер памяти транслирует (отображает) их в физические адреса. ОС предотвращает чтение или изменение одним пользователем памяти, занимаемой другим пользователем (если только эта память явно не объявлена совместно используемой).

Совместимость

Рассмотрим сначала совместимость по программным интерфейсам. Под такой совместимостью понимают способность ОС выполнять программы, написанные для другой ОС или предыдущих версий той же самой системы.

При этом рассматривают два вида совместимости: двоичную совместимость и совместимость на уровне исходных текстов. Если исполняемый файл можно запустить в другой ОС, то достигнута двоичная совместимость. Совместимость на уровне исходных текстов требует предварительной перекомпиляции программы. Если новый процессор использует тот же набор команд и ту же адресацию, что и старый, то будет достигнута двоичная совместимость. Если это не так, то можно добиться двоичной совместимости при помощи программы эмулятора, преобразующего один набор машинных команд в другой. При отсутствии эмулятора все переносимые приложения необходимо заново скомпилировать, скомпоновать и вероятно отладить.

При помощи защищенных подсистем (подсистем окружения) NT предоставляет среду для выполнения приложений, использующих API отличающийся от ее собственного API Win32. Windows NT обеспечивает двоичную совместимость для приложений MS-DOS, 16-разрядной Windows и OS/2. Кроме того, NT обеспечивает совместимость на уровне исходных текстов для приложений POSIX. Мы рассмотрели совместимость по программным интерфейсам.

Рассмотрим совместимость на уровне файловых систем. NT поддерживает ряд существующих файловых систем:

- Файловую систему MS-DOS (FAT);
- Высокопроизводительную файловую систему OS/2 (HPFS);
- Файловую систему для CD-дисков (CDFS);
- Свою новую восстанавливаемую файловую систему (NTFS).

Производительность

Производительность постоянно имелась в виду при разработке всех компонентов системы. Производилось тестирование и моделирование быстродействия компонентов, критичных для производительности. Критические участки вычислений тщательно оптимизировались для достижения максимальной скорости обработки.

Учитывая, что защищенные подсистемы часто взаимодействуют друг с другом и с пользовательскими приложениями. Чтобы это взаимодействие не снижало производительности в состав ОС был включен высокоскоростной механизм передачи сообщений - локальный вызов процедур (*local procedure call*, LPC).

Глава

1

Архитектура Windows

Важнейшая особенность архитектуры Windows - это переносимость между различными аппаратными платформами. Ключевой компонент, обеспечивающий такую переносимость, - это уровень ***аппаратных абстракций*** (***hardware abstraction layer, HAL***).

Уровень аппаратных абстракций (HAL)

HAL - это загружаемый модуль режима ядра (Hal.dll), предоставляющий низкоуровневый интерфейс с аппаратной платформой, на которой выполняется Windows. Он скрывает от ОС специфику конкретной аппаратной платформы, в том числе ее интерфейсов ввода-вывода, контроллеров прерываний и механизмов взаимодействия между процессорами, т.е. все функции, зависимые от архитектуры и от конкретной машины.

Когда внутренним компонентам Windows и драйверам устройств нужна платформенно-зависимая информация, они обращаются не к самому оборудованию, а к подпрограммам HAL, что и обеспечивает переносимость ОС.

Ядро (Kernel)

Ядро (kernel) - компонент исполнительной системы, управляющий процессором. Выполняет планирование и переключение потоков, обработку прерываний и исключений и мультипроцессорную синхронизацию, а также реализует примитивные объекты, используемые исполнительной системой для создания объектов пользовательского режима.

Ядро является основой вычислительной модели и обеспечивает поддержку многозадачности. Оно не определяет определенную политику/стратегию для управления процессами, памятью, файлами и устройствами. Ядро обеспечивает определенный уровень поддержки и его можно рассматривать как набор предлагаемых строительных компонент, который вы можете использовать. Клиенты ядра могут комбинировать эти компоненты для построения более сложных компонент, которые уже будут определять политику/стратегию.

Ядро предоставляет объекты и потоки (абстракции вычислений) на основе HAL и аппаратного обеспечения. Приложения, которое является клиентом ядра, использует эти абстракции для взаимодействия с аппаратурой. Для реализации объектов и потоков ядро управляет аппаратными прерываниями и исключениями, выполняет планирование и мультипроцессорную синхронизацию.

Объекты (Objects)

Ядро содержит набор встроенных типов объектов (классы в объектно-ориентированном программировании). Некоторые типы объектов ядра используются самим ядром. Эти объекты позволяют сохранять и управлять состоянием ядра. Другие объекты используются исполнительной системой, подсистемами и кодом приложений как основа вычислительной модели.

Объекты ядра должны быть быстродействующими. Они существуют в режиме ядра в соответствующем контексте, здесь отсутствует проверка безопасности по сравнению с "нормальными" объектами для которых такая проверка производится. При этом необходимо учитывать, что объектами ядра нельзя управлять из программ пользовательского режима, а только используя вызовы функций. Существует две группы объектов ядра: управляющие объекты (control objects) диспетчерские объекты (dispatcher objects).

Управляющие объекты (Control objects)

Управляющие объекты используются для управления аппаратным обеспечением и другими ресурсами ядра. Когда приложение создает новый процесс, оно требует, чтобы ядро создало управляющий объект процесс. После создания операционная система возвращает приложению *описатель* (*handle*) созданного объекта процесса и приложение ссылается на объект, используя этот описатель. Когда приложение манипулирует процессом, оно манипулирует объектом процессом ядра. Существуют другие управляющие объекты: объект *асинхронный вызов процедуры* (*asynchronous procedure call, APC*), объект

отложенный вызов процедуры (*deferred procedure call, DPC*) и несколько объектов, используемых системой ввода-вывода, в их числе объект прерывание, объект уведомление питания, объект состояние питания.

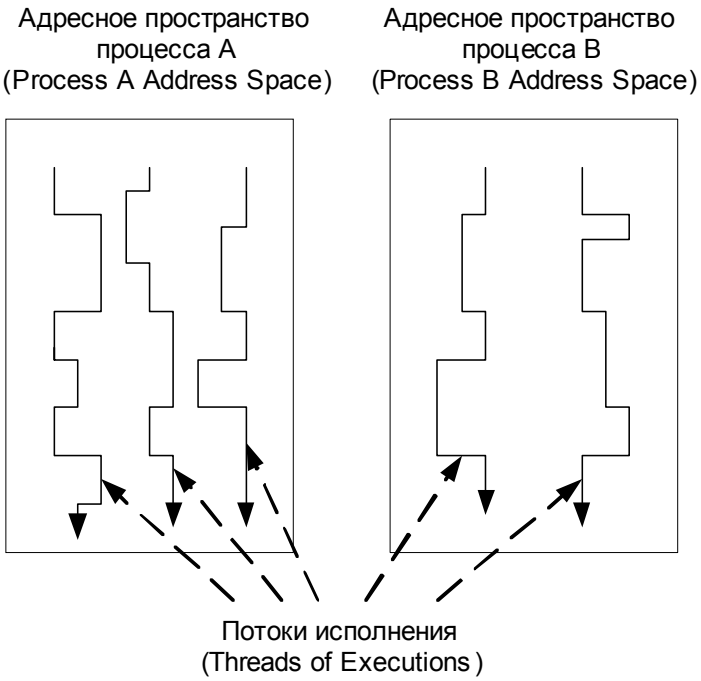
Диспетчерские объекты (Dispatcher objects)

Диспетчерские объекты используются для планирования и синхронизации потоков. Каждый диспетчерский объект имеет встроенные характеристики, которые используются для поддержки синхронизации пользовательского уровня. Объект процесс (управляющий объект) это абстракция вычисления, которая включает адресное пространство и набор ресурсов. Однако в Windows объект процесс не может выполняться. Диспетчерский объект - объект поток это активный элемент, выполняемая вычислительная абстракция. Объект поток имеет свой собственный стек и может выполняться в процессе. Когда любое приложение выполняется оно должно иметь ассоциированный с ним управляющий объект процесс и диспетчерский объект поток. Другие диспетчерские объекты используются для реализации различных форм синхронизации.

Потоки (Threads)

Как уже упоминалось при обсуждении объектов, поток это абстракция вычислений. В Windows объект процесс определяет адресное пространство, в котором один или более потоков могут выполняться. Каждый объект поток представляет отдельную выполняемую сущность внутри процесса. В UNIX может быть только дин поток, выполняемый в каждом адресном пространстве, т.е. существует только один путь выполнения в адресном пространстве процесса. В других современных ОС (Mach, Chorus и NT) можно создать более одного потока (пути выполнения кода) в адресном пространстве процесса. Потоки называют легковесными процессами. Они существуют в адресном пространстве процесса и совместно используют ресурсы (*Рисунок 1.1*).

Рисунок 1.1 Процессы и потоки

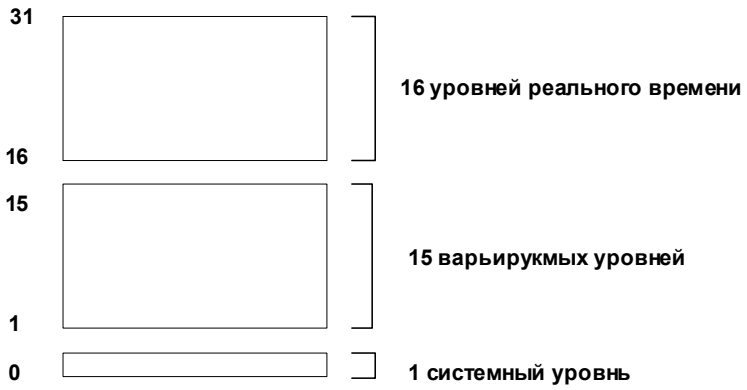


Планирование потоков

В Windows используется приоритетный, с вытеснением и квантованием времени планировщик. Процессор выделяется потоку на квант времени, вычисляемый как несколько тиков системных часов. Планировщик поддерживает 32 уровня приоритета и соответственно столько же различных очередей планировщика. Как и во всех многоуровневых планировщиках, пока существуют потоки в очереди самого высокого приоритета, потоки только из этой очереди будут назначаться на процессор. Если в этой очереди нет больше потоков, тогда планировщик будет обслуживать очередь второго по величине приоритета и т.д. Значения приоритетов группируются так (*Рисунок 1.2*):

- шестнадцать уровней реального времени (16-31 Real-time level);
- пятнадцать варьируемых (динамических) уровней (1-15 Variable-level);
- один системный уровень (0 System-level) зарезервирован для потока обнуления страниц (zero page thread).

Рисунок 1.2 Уровни приоритета



Исполнительная система (Executive System)

Исполнительная система строится на базе ядра и реализует полный набор политик и сервисов Windows, включая управление процессами, памятью, файлами и устройствами. В системе использован объектно-ориентированный подход, поэтому разделение на модули не строго следует классическому подходу разделения на модули в соответствии с функциональностью. Вместо этого исполнительная система спроектирована и реализована как слой, состоящий из набора следующих модулей:

- Диспетчер объектов (Object Manager);
- Диспетчер процессов и потоков (Process and Thread Manager);
- Диспетчер виртуальной памяти (Virtual Memory Manager);
- Справочный монитор защиты (Security Reference Monitor);
- Диспетчер ввода/вывода (I/O Manager);
- Диспетчер кэша (Cache Manager);
- Средство локального вызова процедур (Local Procedure Call).

Диспетчер объектов (Object Manager)

Диспетчер объектов исполнительной системы реализует другую объектную модель на базе объектов ядра (*Рисунок 1.3*). Тогда как, объекты ядра существуют в безопасном окружении, объекты исполнительной системы используются другими частями исполнительной системы и программным обеспечением

пользовательского режима и должны быть приняты серьезные меры по обеспечению их надежности и безопасности. Рассмотрим, как работает диспетчер объектов.

Объект исполнительной системы существует в области ядра, а потоки пользователя должны иметь возможность ссылаться на него. Это осуществляется за счет того, что диспетчер объектов предоставляет *описатель* (*handle*) для каждого объекта исполнительной системы. Когда потоку нужен новый объект исполнительной системы, он вызывает функцию диспетчера объектов для создания объектов (в области ядра), создает описатель объекта (в адресном пространстве процесса) и затем возвращает описатель вызывающему потоку.

Рисунок 1.3 Описатели, объекты исполнительной системы и объекты ядра

Другой поток при желании может использовать уже созданный объект исполнительной системы. Когда этот поток пытается создать уже существующий объект, диспетчер объектов отмечает, что объект уже существует и создает второй описатель для этого потока, который ссылается на тот же самый (уже существующий) объект исполнительной системы. Диспетчер объектов хранит число созданных описателей объекта исполнительной системы. Когда эти описатели

будут уничтожены (закрыты), то будет уничтожен и сам объект. Очень важно, чтобы каждый поток закрывал открытые описатели, когда они больше не нужны.

Существует 15 предопределенных типов объектов. Когда объект создается, он включает заголовок (используется диспетчером объектов для управления объектом) и тела, содержащего зависимою от типа объекта информацию. Заголовок включает:

- **Имя объекта.** Позволяет различным процессам ссылаться на объект.
- **Дескриптор защиты.** Содержит разрешения доступа.
- **Информация об открытых описателях.** Содержит информацию о том, какие процессы используют описатели объекта.
- **Тип объекта.**
- **Счетчик ссылок.** Содержит количество открытых описателей объекта.

Диспетчер объектов работает с заголовком. Например, когда создается новый описатель объекта, диспетчер объектов обновляет информацию об открытых описателях и увеличивает счетчик ссылок. Информация о типе объекта определяет стандартный набор методов объекта (**`open`**, **`close`**, **`delete`**). Некоторые из этих методов поддерживаются диспетчером объектов, тогда как другие зависят от типа объекта, хотя интерфейс описан как часть заголовка объекта.

Формат тела объекта определяется тем компонентом исполнительной системы, который использует объект. Например, если нас интересует объект файл, то формат и содержимое тела будет определяться диспетчером файлов исполнительной системы.

Диспетчер процессов и потоков (Process and Thread Manager)

Диспетчер процессов и потоков (*Process Manager*) служит для тех же целей, что диспетчер процессов в любой ОС. Он отвечает за:

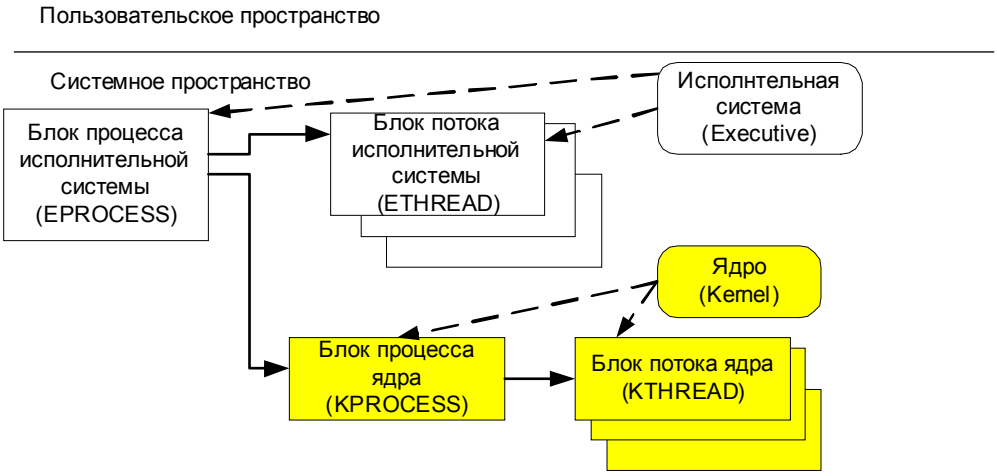
Создание и уничтожение процессов и потоков;

- Распределение ресурсов;
- Поддержку примитивов синхронизации;
- Управление изменением состояния процессов и потоков;
- Сохранение необходимой информации о каждом процессе и потоке.

Диспетчер процессов реализует абстракцию процесс, которую используют подсистемы и приложения. Для этого он определяет структуры данных для хранения состояния каждого процесса и потока (*Рисунок 1.4*). Основной дескриптор процесса называется *executive process control block* (*EPROCESS*).

Он содержит информацию, которую содержит любой дескриптор процесса (идентификатор, список ресурсов, описание адресного пространства). Блок EPROCESS также ссылается на блок управления процессом, который называется KPROCESS и содержит информацию о процессе в области ядра. Ядро манипулирует этой порцией информации в блоке EPROCESS, а исполнительная система отвечает за поддержку остальных полей.

Рисунок 1.4 Дескрипторы процесса и потока



Учитывая близкие отношения между процессом и потоком, точно также существует *executive thread process control block* (*ETHREAD*) для каждого потока процесса. И блок EPROCESS ссылается на список блоков ETHREAD. Диспетчер процессов использует информацию в блоке ETHREAD для управления потоком. А так как поток строится на базе потока уровня ядра, также существует kernel *thread control block* (*KTHREAD*). Он содержит информацию о объекте потоке ядра, которым управляет ядро. Блок EPROCESS ссылается на блок KPROCESS, который в свою очередь ссылается на набор блоков KTHREAD.

Функция ядра NtCreateProcess вызывается, когда вызывается функция Win32 API CreateProcess. Когда это происходит, то выполняется следующая работа:

- Вызывается ядро для создания объекта процесса ядра;

- Создается и инициализируется блок EPROCESS;
- Создается адресное пространство процесса.

Процесс не может выполнять код в своем адресном пространстве. Для этого ему необходим, по крайней мере один поток управления, который называется базовым потоком. Функция NtCreateThread создает поток, который будет выполняться внутри процесса. Вызов функции Win32 API CreateProcess приводит к вызову функций NtCreateProcess и NtCreateThread. Для создания дополнительных потоков можно использовать CreateThread при этом будет вызвана функция ядра NtCreateThread. При этом выполняется следующая работа:

- Вызывается ядро для создания объекта потока ядра;
- Создается и инициализируется блок ETHREAD;
- Инициализируется поток для выполнения;
- Поток размещается в очереди планировщика.

Диспетчер виртуальной памяти (Virtual Memory Manager)

Windows - это система со страничной виртуальной памятью. В такой системе содержимое адресного пространства процесса хранится во вторичной памяти и при необходимости происходит постраничная загрузка из вторичной в первичную (выполняемую) память.

При создании процессу выделяется виртуальное адресное пространство 4 Гб, хотя в этот момент ни один из адресов в действительности не размещен. Когда процесс нуждается в памяти, то во-первых необходимое адресное пространство ***резервируется*** для последующего использования (при этом в действительности размещения в памяти не происходит). Когда процессу нужно сохранить информацию, он ***передает*** адресное пространство, при этом процессу действительно предоставляется память для хранения информации. Обычно операция передачи приводит к тому, что область на диске (в страничном файле) передается процессу. Информация хранится на диске до тех пор, пока на нее действительно не ссылается поток.

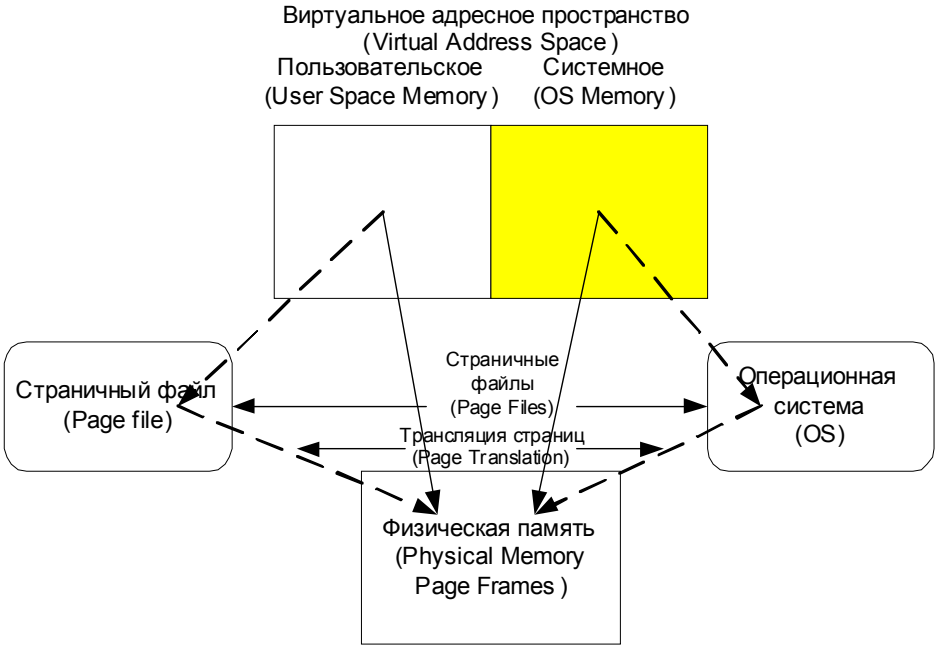
Подобно всем механизмам виртуальной памяти, когда выполняемый поток ссылается на виртуальный адрес, диспетчер виртуальной памяти гарантирует, что страница, содержащая виртуальный адрес, будет считана из страничного файла и размещена в физической памяти.

Диспетчер виртуальной памяти отобразит виртуальный адрес, на который ссылается поток, в физический адрес, где размещена информация (*Рисунок 1.5*).

Диспетчер виртуальной памяти спроектирован так, что большая часть адресного пространства каждого процесса (обычно половина) отображается на область, используемую системой в режиме ядра. Для этого существует несколько важных соображений:

- Процесс может напрямую ссылаться на любое место в системной области;
- Все процессы разделяют одну системную область;
- Такое большое совместно используемое адресное пространство реализуется за счет отображаемых в память файлов.

Рисунок 1.5 Виртуальная память



Когда поток ссылается на адрес в пользовательском адресном пространстве, система виртуальной памяти загружает требуемую область в физическую память, чтобы поток мог читать или писать по этому виртуальному адресу, ссылаясь на соответствующий физический адрес. Такое же отображение имеет место и для системного адресного пространства, но эти ссылки защищены. Системная часть адресного пространства любого процесса отображается на общую системную область, в отличие от собственной пользовательской части приложения.

Справочный монитор защиты (Security Reference Manager)

Ядро поддерживает низкоуровневые механизмы аутентификации. На уровне исполнительной системы за политики защиты отвечает ***справочный монитор защиты*** (***Security Reference Manager***). Он контролирует возможность доступа к объектам в соответствии с заданной политикой (политика задается на уровне подсистем).

Механизмы защиты, реализуемые справочным монитором защиты, используются вместе с модулем защиты пользовательского режима. Этот модуль называется *подсистемой защиты* (*Local Security Authority, LSA*) и позволяет устанавливать необходимую политику защиты. Подсистема защиты использует базу данных безопасности, которая хранится в реестре и определяет политику компьютера. Политику компьютера можно изменить, редактируя эту базу данных. При аутентификации сервер LSA сравнивает запрос с содержимым базы данных.

Справочный монитор защиты производит аутентификацию доступа к объектам исполнительной системы. Когда любой поток выполняет системный вызов для определенного доступа к объекту исполнительной системы, запрос на именно такой доступ передается справочному монитору защиты. Объект содержит дескриптор безопасности, который содержит владельца объекта и список контроля доступа (access control list, ACL) процессов, которым разрешен доступ к объекту. Справочный монитор защиты определяет идентификатор потока и тип требуемого им доступа, а затем проверяет, разрешен ли ему заданный доступ (используя информацию в ACL).

Подсистемы (Subsystems)

Серверы Windows называются ***защищенными подсистемами*** (***protected subsystems***), так как каждый из них - это отдельный процесс, память которого защищена от других процессов системой виртуальной памяти. Термин "сервер" подразумевает, что каждая защищенная подсистема обеспечивает API, который могут использовать программы. Когда приложение (или другой сервер) вызывает некоторую процедуру API, серверу, реализующему данную процедуру, посылается сообщение при помощи средства локального вызова процедур (local procedure call, LPC) - оптимизированный механизм исполнительной системы для локальной передачи сообщений. Сервер посылает ответное сообщение вызывающей программе.

В Windows имеется два типа защищенных подсистем: ***подсистемы среды*** (***environment subsystems***) и ***неотъемлемые подсистемы*** (***integral subsystems***). Подсистема среды это сервер пользовательского режима, реализующий API некоторой ОС. Когда приложение вызывает функцию API, этот вызов доставляется посредством LPC подсистеме среды. Та исполняет вызов и возвращает результаты прикладному процессу, посылая другой LPC.

Самая важная подсистема среды в Windows это подсистема Win32, которая предоставляет прикладным программам Win32 API, кроме того, она реализует графический интерфейс пользователя и управляет всем вводом/выводом приложений.

Неотъемлемая подсистема - это подсистема защиты. Под Win32 API мы будем понимать базовый набор функций, предназначенных для поддержки процессов, потоков, управления памятью, защиты, ввода-вывода, операций с окнами, графики и др. (см. *Microsoft Developer Network*, MSDN).

Win32 API

Это основной интерфейс программирования в семействе ОС Microsoft Windows, включая Windows 2000, Windows 95, Windows 98, Windows Millennium Edition и Windows CE.

Под Win32 API мы будем понимать базовый набор функций, предназначенных для поддержки процессов, потоков, управления памятью, защиты, ввода-вывода, операций с окнами, графики и др. (см. Microsoft Developer Network, MSDN).

ЛАБОРАТОРНАЯ РАБОТА 1

ВВОДНАЯ

Теория

Практически все современные компьютеры используют архитектуру фон Неймана. Идея заключается в том, что вычисления будут выполнены, если программа будет находиться в оперативной памяти, а устройство управления будет извлекать и декодировать инструкции из памяти. Устройство управления далее взаимодействует с другими аппаратными компонентами для выполнения инструкции. На этом уровне вычислений не существует понятий пользователей, процессов или разделяемых ресурсов, а просто устройство управления выполняет одну за другой инструкции из памяти.

Многозадачная ОС создает виртуальную машину - абстракцию. В этом случае несколько программ должны одновременно находиться в оперативной памяти, а устройство управления периодически переключается с выполнения инструкций одной программы на выполнения инструкций другой. На аппаратном уровне это не имеет значения, что выполняются инструкции разных программ. Для устройства управления такое переключение это просто выполнение инструкции перехода. Однако с точки зрения пользователей это означает, что компьютер выполняет одну программу некоторое время, а потом переключается на выполнение другой (процессор это разделяемый ресурс). ОС реализует этот мультипрограммный подход, давая программистам возможность загружать программы в память, обеспечивая их выполнение, приостанавливая выполнение и т.д. Построение такого программного окружения это результат определенной стратегии, которая называется *software abstractions*. Так *ядро* создает абстракции объектов потоков на базе архитектуры фон Неймана. *Исполнительная система* использует абстракции ядра для создания модели *процесса*, в котором может исполняться поток. Поток не может выполнять программу, которая не находится в адресном пространстве его процесса, а потоки других процессов не могут выполняться в его адресном пространстве. В этом упражнение вы познакомитесь с процессами и потоками.

Task Manager (*TASKMGR.EXE*)

Для запуска щелкнуть правой кнопкой мыши на панели задач (в той области, которая не занята кнопками приложений), а затем выбрать в появившемся контекстном меню элемент *Task Manage*r. Появится окно *Windows* NT *Task Manager*. По умолчанию, в окне *Task Manager* открыта вкладка *Applications*. Имеются еще вкладки Processes и Performance.

Вкладка *Applications* (Приложения)

В **Applications** *о*тображается список работающих в системе риложений. Эта вкладка используется главным образом для того, ать зависшие приложения (**End Task**). Можно переключиться на выделенное приложение (**Switch On**) или запустить новое приложение *(***New Task***).*

Вкладка Processes(Процессы)

На этой вкладке приведен перечень всех процессов, выполняемых в системе. Это более длинный и подробный список, нежели приведенный на вкладке *Applications*. Здесь указывается имя процесса, идентификатор процесса, процессорное время (в процентах от общего времени) и объем памяти, используемый каждым процессом. Используя эту информацию, вы можете выявить "пожирателей" памяти. *Task Manage*r помогает справиться со случайными проблемами, вызванными одним или несколькими невидимыми - и зависшими - экземплярами приложения (часто Microsoft Internet Explorer*)*. Если пользователь жалуется, что программа не запускается, сколько бы он не щелкал на элементе или пиктограмме меню *Start,* проверьте информацию в *Task Manager/Processes* и поищите соответствующий exe-файл. Если найдете, остановите программу и все копии кнопкой *End Process*, и пользователь наверняка сможет работать дальше.

Вкладка Performance (Производительность)

Эта вкладка содержит сведения о памяти, а также графическое представление информации об использовании ресурсов процессора и памяти. Почти такие же сведения предоставляют объекты *Processor* и *Memory* в *Performance Monitor*, но гораздо проще посмотреть их графическое отображение здесь. Эта графическая информация позволяет сразу же увидеть, вызвана ли избыточная активность диска пробуксовкой виртуальной памяти (счетчик *Commit Change Total* почти достиг предельного значения, а степень использования центрального процессора неизменно высока). В этом случае необходимо изменить параметры виртуальной памяти. Кроме того, *Task Manager* может помочь вам обнаружить пожирателя ресурсов процессора. Если пользователь жалуется, что его система, работает слишком медленно, сначала посмотрите на вкладку *Performance*.

Process Viewer (PVIEW.EXE)

Process Viewer - это часть окружения *Visual* C++. Вы можете вызвать его, набрав полное путевое имя, где у вас находится *PVIEW.EXE* в cmd.exe или вызвать его из "*Start/Programs/Visual C++*" меню. *Process Viewer* - это графическое приложение Windows, которое выводит информацию о состоянии системы. Например, имя вашего компьютера. Кроме того, оно выводит список активных процессов, это теже самые процессы, которые *Task Manager* отображает в окне *Processes*. Дополнительно выводится информация, какое время процесс выполняется в режиме ядра, а какое время в пользовательском режиме. Первая строка "*_Total*" - это суммарное время.

Process Viewer выводит информацию, которая используется при планировании и диспетчеризации потоков. В поле "*Priority*" выводится класс приоритета процесса, а в поле "*Thread Priority*" - приоритет потока. Выбрав для просмотра процесс, вы получите список всех его потоков. Далее выбрав поток можно получить детальную информацию о нем (стартовый адрес, сколько раз происходило переключение контекста, динамический приоритет).

Process Viewer дает снимок состояния системы при запуске, и вы должны использовать *Refresh* для получения нового состояния.

Некоторые системные процессы

В *Windows* на каждой машине выполняется множество различных процессов. Рассмотрим некоторые из них.

System Idle Process. В этом процессе существует только один поток. Он присутствует в каждой системе. Выполняется в случае, если больше ничто не выполняется. Этот поток процесса называют еще потоком процесса "обнуления страниц".

System. Процесс *System* служит носителем особых потоков, работающих только в режиме ядра, - *системных потоков режима ядра* (*kernel-mode system threads*), которые выполняют различные функции ОС. Этот процесс используется различными частями NTOSKRNL или драйверами. В этом процессе запущено много потоков.

Session Manager (***SMSS.EXE***). Этот процесс инициализирует различные части исполнительной системы и подсистемы Win32. А после завершения инициализации обеспечивает взаимодействие между приложениями и отладчиком.

Win 32 Subsystem (***CSRSS.EXE***). Подсистема Win32 управляет дисплеем и клавиатурой, а также выполняет другую работу по поддержке Win32 API.

Windows Logon (***WINLOGIN.EXE***). Когда пользователь собирается начать или закончить работу в системе, этот процесс позволяет это сделать. В результате успешной регистрации пользователя этот процесс создает процесс USERINIT.EXE. Этот процесс, в свою очередь, запускает другой процесс EXPLORER.EXE для управления взаимодействием между пользователем и компьютером. После этого USERINIT.EXE заканчивается.

Local Security Authority Server (**LSASS.EXE**). Во время аутентификации пользователя процесс WINLOGIN взаимодействует с сервером *LSA* или с *LSASS* процессом. Он выполняет пользовательскую часть процедуры аутентификации для доступа к объектам, взаимодействуя с *Executive Security Reference Monitor.*

Services (***SERVICES.EXE***). Система предоставляет сервисы, которые могут быть драйверами или некоторыми системными процессами. Примерами сервисов являются части ОС, которые управляют спулингом при печати или сетевыми возможностями. *SERVICES* создают другие процессы в зависимости от того, как сконфигурирована система.

Чему нужно научиться ?

У вас должно появиться интуитивное представление о том, что такое процессы и потоки.

Задание

1. Проверьте, что все приложения закрыты.
2. Запустите cmd.exe *("Start/Programs/Command prompt*").
3. Наберите команду *hostname*, чтобы узнать имя вашего компьютера.
4. Наберите *winmsd.exe (Windows NT Diagnostics)* или "*Start/Programs/ Administrative Tools/ Windows* NT Diagnostics".
   - Определите версию системы;
   - Какой тип CPU используется;
   - Сколько физической памяти;
   - Сколько памяти используется ядром;
   - Сколько сервисов запущено.

5. Закройте winmsd.exe.

6. Запустите pview.exe. Определите следующие характеристики:

   - Сколько процессов запущено;
   - Запущен ли процесс *Idle;*
   - Сколько времени процесс *Idle* проводит в режиме ядра, а сколько в пользовательском режиме;
   - Какой класс приоритета у процесса *Idle;*
   - Какой приоритет у потока процесса *Idle;*
   - Какой динамический приоритет у потока процесса *Idle;*
   - Найдите в списке процесс pview;
   - Какой класс приоритета у процесса pview;
   - Сколько потоков запущено в процессе pview;
   - Какой приоритет у потоков процесса pview;
   - Какой динамический приоритет у каждого потока процесса pview;
   - Есть ли процессы в которых запущено больше чем два потока.

7. Запустите *Task Manager*

   - Какие приложения запущены;
   - Посмотрите на вкладку *Processes* (*Task Manager*). Обратите внимание на поле PID (идентификатор процесса) и на номер в скобках в pview. Есть ли между ними связь;
   - Сколько описателей открыто в системе;
   - Сколько описателей открыто в системе;
   - Сколько процессов существует в системе;
   - Сколько потоков существует в системе.

8. **Изменение приоритетов приложений**.

Для просмотра приоритета запущенного приложения:

- Start -> Programs -> Accessories -> Clock;
- Запустите второй экземпляр *Clock;*
- Найдите в *Task Manager* два экземпляра *Clock;*
- Выберите вкладку *Processes;*
- В меню *View* выберите *Select Columns;*
- Включите *Base Priority* и нажмите **OK**
- Если необходимо, увеличьте окно, чтобы увидеть колонку *Base Priority;*

- С каким приоритетом запущены *Clock;*

Для изменения приоритета запущенного приложения:

- Укажите на ***Set Priority*** и нажмите **Low**. Появится диалоговое окно предупреждения (*Task Manager Warning*);
- Чтобы изменить приоритет приложения, нажмите **Yes.** Низкоприоритетный экземпляр приложения будет работать значительно медленнее, чем экземпляр с нормальным приоритетом
- В ***Task Manager*** вкладка ***Processes*** выберите ***Clock*** (***Low***) и, нажимая правую клавишу мыши, вызовите контекстное меню. Для этого выполните следующие шаги:

  •·Укажите на ***Set Priority*** и нажмите **High**. Появится диалоговое окно предупреждения (*Task Manager Warning*).

  •·Чтобы изменить приоритет приложения, нажмите **Yes**. Высокоприоритетный экземпляр приложения будет работать значительно быстрее, чем экземпляр с нормальным приоритетом.

  •·В ***Task Manager*** вкладка ***Processes*** выберите ***Clock*** (***High***) и, нажимая правую клавишу мыши, вызовите контекстное меню.

  •·Укажите на ***Set Priority*** и нажмите **Real Time**. Появится диалоговое окно предупреждения (*Task Manager Warning).*

  •·Чтобы изменить приоритет приложения, нажмите **Yes**. Экземпляр приложения с приоритетом реального времени монополизирует процессор.

  •·Перезагрузитесь (***Restart***).

Глава

2

Процессы и потоки

В однозадачной ОС пользователь может запустить одновременно не более одной программы и только после того как она закончит работу можно будет запустить другую. В многозадачной ОС пользователи могут одновременно запускать несколько программ или даже несколько копий одной программы.

В чем разница между программой и процессом? Программа - это статическая последовательность команд, а ***процесс*** (***process***) - это программа и системные ресурсы, необходимые для ее выполнения. Процесс является субъектом владения ресурсами и единицей работы. ОС выделяет каждому процессу порцию системных ресурсов и гарантирует, что программа каждого процесса будет направляться на исполнение в определенном порядке и своевременно.

ОС содержит блок кода, управляющий созданием и удалением процессов, а также отношениями между ними. Этот код называется ***структурой процессов*** (***process structure***) и в Windows NT реализован ***диспетчером процессов*** (***process manager***). Его основная задача предоставить набор базовых сервисов процесса, которые подсистемы среды могли бы использовать для эмуляции своих собственных уникальных структур процессов.

В разных ОС процессы реализованы по-разному. Они различаются своим представлением (структурами данных), способами именования и защиты, а также отношениями между собой. Базовые процессы Windows NT имеют ряд характеристик, отличающих их от процессов других ОС:

- процессы реализованы как объекты, и доступ к ним осуществляется посредством объектных сервисов;
- в адресном пространстве процесса может исполняться несколько потоков;

- объект-процесс и объект-поток имеют встроенные возможности синхронизации.

Что такое процесс?

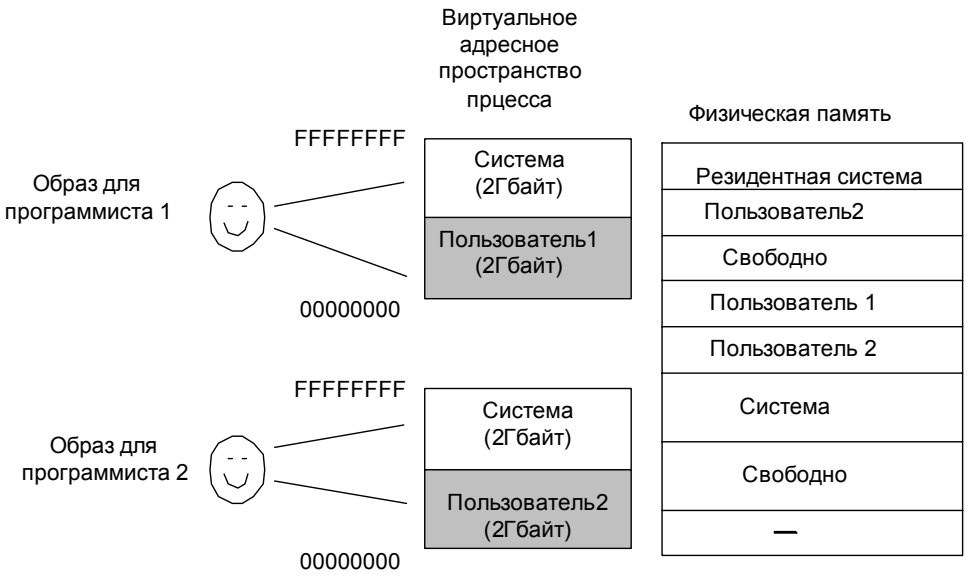
Процесс состоит из:

- исполняемой программы (код и данные);
- закрытого ***адресного пространства*** (***address space***), т.е. набора адресов виртуальной памяти, который процесс может использовать;
- системных ресурсов, выделяемых ОС процессу во время выполнения программы (семафоров, файлов и т.д.);
- по крайней мере, одного ***потока управления*** (***thread of execution***). ***Поток*** - это сущность внутри процесса, которую ядро NT направляет на исполнение. Без него программа процесса не может выполняться.

Адресное пространство

С помощью системы ***виртуальной памяти*** (***virtual memory***) программисты (и создаваемые ими процессы) получают логический образ памяти, который не совпадает с ее физической структурой (*Рисунок 2.1*).

Рисунок 2.1 Виртуальная и физическая память



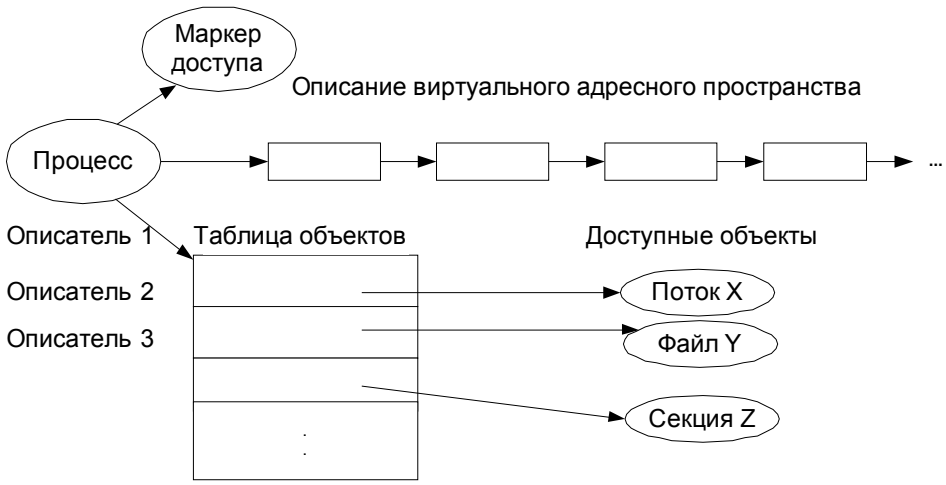
При всяком обращении процесса по виртуальному адресу система виртуальной памяти транслирует этот адрес в физический адрес. Она также предотвращает непосредственный доступ процесса к виртуальной памяти, занятой другими процессами или ОС. Для исполнения кода ОС или доступа к памяти ОС поток должен выполняться в ***режиме ядра*** (***kernel mode***). Большинство процессов - это процессы ***пользовательского режима*** (***user mode***).

Поток пользовательского режима получает доступ к ОС, вызывая некоторый системный сервис. Когда поток вызывает сервис, происходит переключение из пользовательского режима в режим ядра. ОС проверяет аргументы, переданные сервису, после чего исполняет сервис. Перед возвратом управления пользовательской программе ОС переключает поток обратно в пользовательский режим.

Системные ресурсы

Кроме закрытого адресного пространства, с каждым процессом связан набор системных ресурсов (*Рисунок 2.2*).

Рисунок 2.2 Процесс и его ресурсы



Маркер доступа (***Access Token***) присоединяет к процессу ОС. Это объект исполнительной системы, который содержит информацию о правах зарегистрированного в системе пользователя, которого представляет данный процесс. Если процессу требуется получить информацию о своем маркере доступа или изменить некоторые атрибуты маркера, он должен открыть описатель своего объекта-маркера. Подсистема защиты определяет, есть ли у объекта такое право.

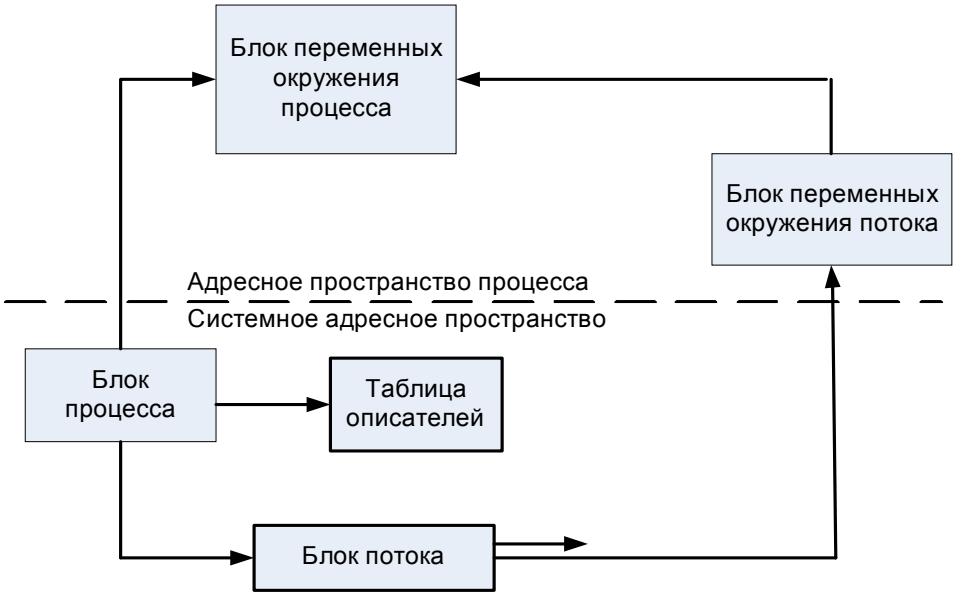
Ниже маркера доступа расположен набор структур данных, созданный диспетчером виртуальной памяти для отслеживания виртуальных адресов, используемых процессом.

Таблица объектов показана внизу. Процесс открыл описатели потока, файла и секции совместно используемой памяти. Описание виртуального адресного пространства содержит информацию о виртуальных адресах, занятых стеком потока и объектом-секцией.

Объект-процесс

Каждый процесс в Windows NT представлен ***блоком процесса***, создаваемым ***исполнительной системой*** (***EPROCESS***). В блоке EPROCESS содержатся атрибуты процесса и указатели на некоторые структуры данных. Так, у каждого процесса есть один или более потоков, представляемых ***блоками потоков*** исполнительной системы (***ETHREAD***). Блок EPROCESS и связанные с ним структуры данных хранятся в системном пространстве. Исключение составляет только ***блок переменных окружения процесса*** (***process environment block, PEB***), он находится в адресном пространстве процесса (*Рисунок 2.3*).

Рисунок 2.3 Блоки переменных окружения процесса (PEB) и потока (TEB)



В исполнительной системе процессы - это объекты исполнительной системы, создаваемые и уничтожаемые диспетчером объектов. Объект-процесс, как и другие объекты, содержит заголовок, создаваемый и

инициализируемый диспетчером объектов. В заголовке хранятся стандартные атрибуты объекта, такие как дескриптор защиты объекта, имя и каталог объектов, в котором хранится имя, если оно есть.

Диспетчер процессов определяет атрибуты, хранящиеся в теле объектов-процессов, а также предоставляет системные сервисы для чтения и изменения этих атрибутов. Атрибуты и сервисы для объектов-процессов показаны на рисунке *(Рисунок 2.4*). Объект процесс исполнительной системы включает объект процесс ядра (содержит указатель на объект процесс ядра). Ядро управляет объектом процесс ядра, а исполнительная система управляет объектом исполнительной системы.

Мы рассматриваем основные атрибуты этих объектов по отдельности, для того чтобы лучше понять какие задачи решаются ядром, а какие исполнительной системой. При выполнении лабораторных работ эта детализация не существенна и надо будет использовать Win32 API функции для работы с объектом процесс исполнительной системы.

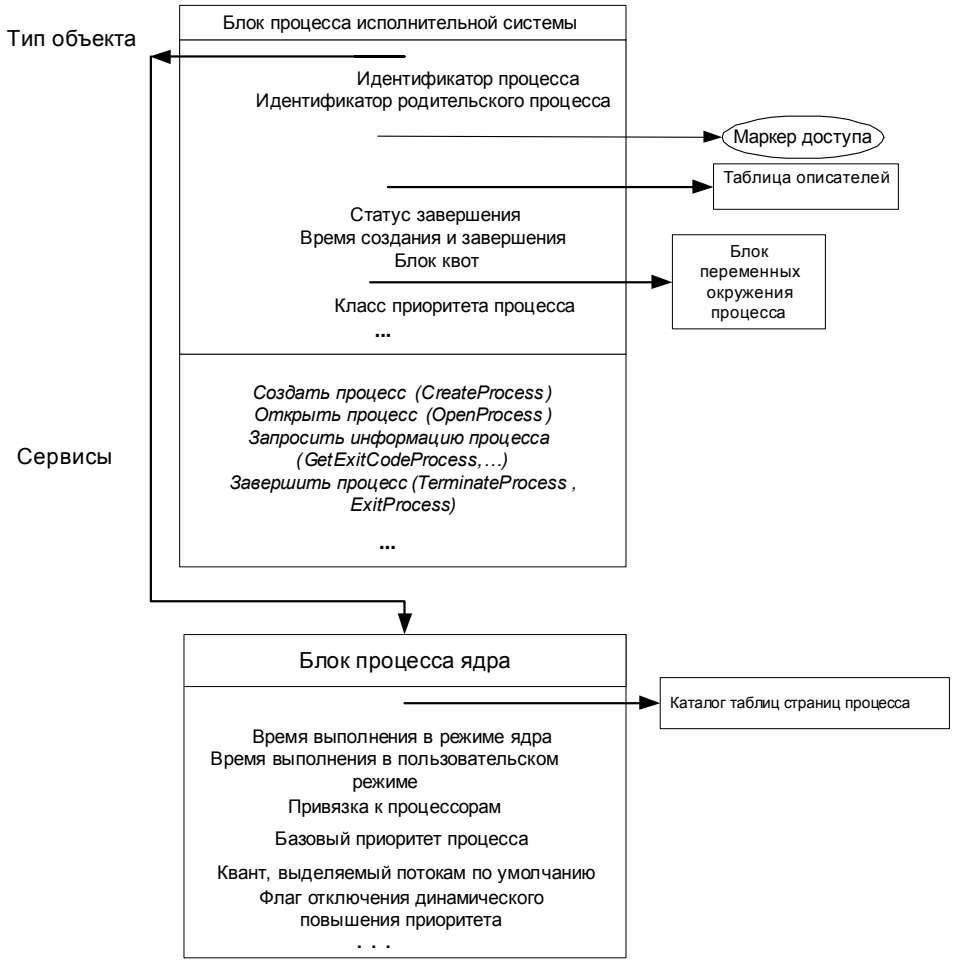
Рассмотрим основные атрибуты:

- *идентификатор процесса* - уникальное значение, идентифицирующее процесс в ОС;
- *базовый приоритет* - базовый приоритет потоков процесса;
- *привязка к процессорам* (*процессорное сродство*) - набор процессоров, на которых потоки процесса могут исполняться по умолчанию;
- *размеры квот* - максимальный объем резидентной и нерезидентной системной памяти, пространства в файле подкачки и процессорного времени, выделяемый пользовательскому процессу;
- *статус завершения* - причина завершения процесса.

Некоторые атрибуты объекта-процесса налагают ограничения на потоки, исполняемые внутри процесса. Например, на многопроцессорном компьютере процессорное сродство может ограничить исполнение потоков процесса только на заданных процессорах. Аналогично, размеры квот регулируют, сколько памяти, пространства в файле подкачки и процессорного времени могут использовать все потоки процесса вместе.

Базовый приоритет процесса помогает ядру NT регулировать приоритет потоков в системе. Приоритеты потоков изменяются, но всегда остаются в диапазоне базовых приоритетов их процессов.

Рисунок 2.4 Блоки процесса исполнительной системы (EPROCESS) и ядра (KPROCESS)



Большинство сервисов объекта-процесса мы будем использовать при выполнении лабораторных работ. Например, сервис завершения процесса останавливает исполнение всех его потоков, закрывает все открытые описатели объектов и уничтожает виртуальное адресное пространство процесс

Что такое поток?

Поток - это единица исполнения, отдельный счетчик команд или подлежащая планированию сущность внутри процесса.

В то время как процесс - это логическое представление работы, которую должна выполнить ОС, поток отображает одну из, возможно, многих необходимых подзадач. Предположим, что пользователь запустил приложение для работы с базой данных. ОС представляет этот вызов

приложения как один процесс. Пусть теперь пользователь запросил генерацию отчета по данным из базы и сохранение этого отчета в файле. Пока идет выполнение этой длительной операции, пользователь ввел новый запрос к базе данных. ОС представляет каждый из запросов - генерацию отчета и новый запрос к базе - как отдельные потоки внутри процесса приложения для работы с базой данных. Эти потоки могут выполняться процессором независимо друг от друга, т.е. обе операции можно выполнять в одно и то же время (параллельно).

Основные составляющие потока в исполнительной системе NT:

- Уникальный идентификатор, называемый идентификатором клиента
- Содержимое набора регистров, отражающее состояние процессора
- Два стека: один используется потоком при работе в пользовательском режиме, а другой - в режиме ядра
- Собственная область памяти, предназначенная для использования подсистемами, библиотеками периода выполнения и динамически подключаемыми библиотеками (DLL).

Регистры, стек, и собственная область памяти называются ***контекстом*** (***context***) потока. Фактически данные, составляющие контекст потока, определяются типом процессора.

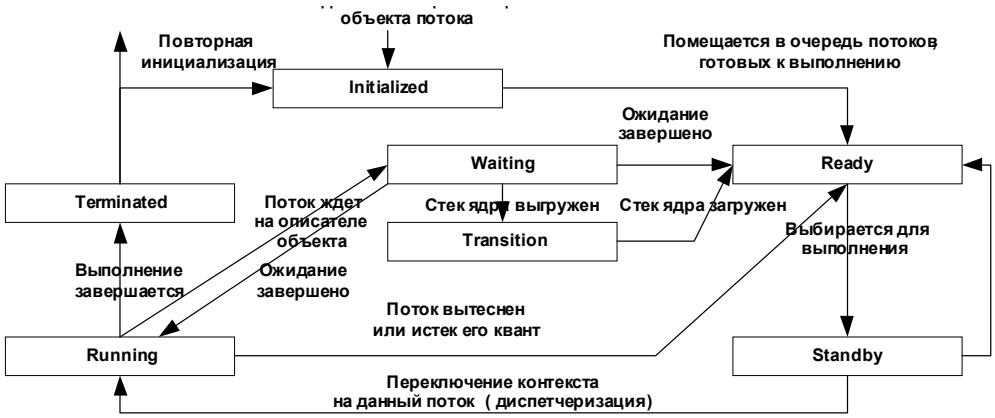
Поток находится в адресном пространстве процесса, используя его для хранения данных во время выполнения. Если в одном процессе существует несколько потоков, то они совместно используют адресное пространство и все ресурсы, включая маркер доступа, базовый приоритет и описатели объектов из таблицы объектов процесса. Ядро NT направляет потоки на исполнение некоторому процессору. Таким образом, каждый процесс NT должен иметь, по меньшей мере, один поток.

Многозадачность и многопроцессорная обработка

ОС с вытесняющей многозадачностью должна использовать тот или иной алгоритм, позволяющий ей распределять процессорное время между потоками. Каждые 20 мс Windows просматривает все существующие объекты потоки и отмечает те из них, которые могут получить процессорное время. Далее она выбирает один из таких объектов и загружает в регистры процессора значение его контекста. Эта операция называется ***переключением контекста*** (***context switching***). Поток выполняет код и манипулирует данными в адресном

пространстве своего процесса. Примерно через 20 мс Windows сохранит значения регистров процессора в контексте потока и приостановит его выполнение. Далее система просмотрит остальные объекты потоки, подлежащие выполнению, выберет один из них, загрузит его контекст в регистры процессора, и все повторится. Этот цикл операций - выбор потока, загрузка его контекста, выполнение и сохранение контекста - начинается с момента запуска системы и продолжается до ее выключения (*Рисунок 2.5*).

Рисунок 2.5 Состояния потоков



Система планирует выполнение только тех потоков, которые могут получить процессорное время. У некоторых объектов-потоков значение ***счетчика простоев*** (***suspend count***) больше 0, это значит, что соответствующие потоки приостановлены и не получают процессорного времени. Кроме приостановленных, существуют и другие потоки, не участвующие в распределении процессорного времени, - они ожидают каких-либо событий.

Поток получает доступ к процессору на 20 мс, после чего планировщик переключает процессор на выполнение другого потока. Но так происходит, только если у всех потоков один приоритет. На самом деле в системе существуют потоки с разными приоритетами, а это меняет порядок распределения процессорного времени.

Каждому потоку присваивается уровень приоритета - от 0 (самый низкий) до 31 (самый высокий). Решая, какому потоку выделить процессорное время, система сначала рассматривает только потоки с приоритетом 31 и подключает их к процессору по принципу карусели. Если поток с приоритетом 31 не исключен из планирования, он получает квант времени, по истечении которого система проверяет, есть ли еще один такой поток. Если есть, то и он получает свой квант процессорного времени.

Пока в системе имеются планируемые потоки с приоритетом 31, ни один поток с более низким приоритетом процессорного времени не получит. Такая ситуация называется ***голоданием*** (***starvation***). Она наблюдается, когда потоки с более высоким приоритетом так интенсивно используют процессор, что остальным ничего не достается.

Кроме того, потоки с более высоким приоритетом вытесняют потоки с более низким приоритетом. Допустим, процессор исполняет поток с приоритетом 5, и тут система обнаруживает, что поток с более высоким приоритетом готов к выполнению. Тогда система остановит поток с более низким приоритетом - даже если не истек отведенный ему квант процессорного времени - подключит к процессору поток с более высоким приоритетом и выдаст ему полный квант времени.

Процессор может выполнять не более одного потока одновременно. Однако ***многозадачная*** (***multitasking***) ***ОС*** дает пользователю возможность исполнять несколько программ, причем создается впечатление, что все они исполняются одновременно. Это достигается следующим образом:

- поток исполняется до тех пор, пока его исполнение не будет прервано или ему не придется ждать освобождения некоторого ресурса;
- сохраняется контекст потока;
- загружается контекст другого потока;
- этот цикл повторяется до тех пор, пока есть потоки, ожидающие выполнения.

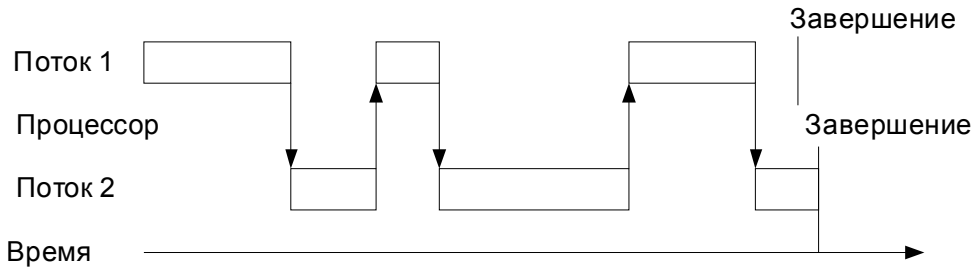
Более детально это изображено на рисунке (*Рисунок 2.5*).

Переключение процессора с исполнения одного потока на исполнение другого потока называется ***переключением контекста*** (***context switching***). В Windows NT оно осуществляется ядром.

Как показано на примере двух потоков (*Рисунок 2.6*), многозадачность, ОС поочередно выполняет то один поток, то другой. В конце концов, каждый поток заканчивает выполнение своей подзадачи и завершается. Очень высокая скорость работы процессора обеспечивает иллюзию одновременного выполнения всех потоков.

Многозадачность увеличивает объем работы, выполняемой системой, потому что большинство потоков не могут исполняться непрерывно. Периодически поток прекращает выполнение и ждет пока другой поток, например, освободит ресурс необходимый первому. Благодаря многозадачности, пока один поток ожидает, может выполняться другой поток, и время процессора не расходуется впустую.

Рисунок 2.6 Многозадачность

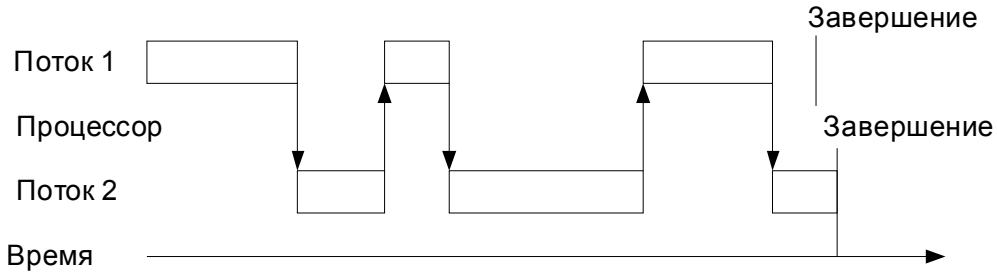


Вытесняющая многозадачность (***preemptive multitasking***) - это разновидность многозадачности, при которой ОС не ждет, пока поток добровольно предоставит процессор другим потокам. Вместо этого ОС прерывает поток, после того как он выполнялся в течение заранее заданного периода времени, так называемого ***кванта времени*** (***time quantum***), или когда готов к выполнению поток с большим приоритетом. Вытеснение предотвращает монополизацию процессора одним потоком и предоставляет другим потокам их долю процессорного времени. Windows NT - это система с вытесняющей многозадачностью. В невытесняющих системах, поток должен был добровольно передавать управление процессором. Плохие программы могут захватить процессор и нарушить работу других приложений или всей системы.

Во многих ОС программы могут иметь только один поток. Фактически в большинстве ОС для обозначения исполняемой сущности используется слово ***процесс*** (***process***).

Поток (***thread***) - новый термин. Так как у каждого процесса имеется отдельное адресное пространство, двум процессам для обмена данными друг с другом нужны специальные средства. Например, для связи между процессами используются каналы (*Рисунок 2.7*).

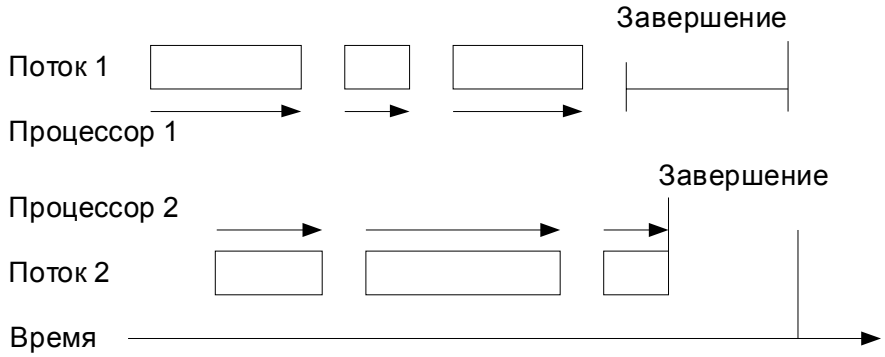
Рисунок 2.7 Компилятор, состоящий из двух процессов



Предпроцессорная обработка и компиляция программы двумя процессами (каждый с одним потоком), будет быстрее, чем в случае одного процесса, так как многозадачная ОС может попеременно исполнять то поток препроцессора, то поток компилятора. Как только препроцессор поместит что-нибудь в совместно используемый буфер, компилятор может начать свою работу. Приложения, которые исполняются в двух или более местах одновременно, называются ***параллельными приложениями*** (***concurrent applications***).

ОС мультипроцессорной обработкой специально спроектирована для работы на компьютерах с несколькими процессорами. ОС с ***симметричной мультипроцессорной обработкой*** (***symmetric multiprocessing SMP***), такая как Windows NT, может выполнять на любом процессоре, как пользовательский код, так и код ОС. Если число потоков превышает число процессоров, ОС SMP также поддерживает многозадачность, разделяя время каждого процессора между всеми ожидающими потоками (*Рисунок 2.8*).

Рисунок 2.8 Мультипроцессорная обработка



Многопоточность

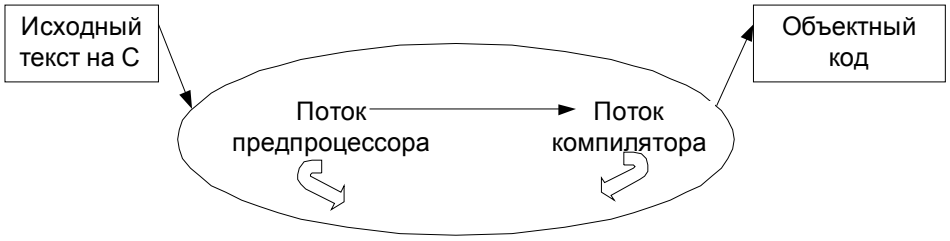
Использование двух процессов для достижения параллельности не всегда эффективно. В некоторых системах UNIX, когда один процесс создает другой, система должна скопировать все содержимое адресного пространства первого процесса в адресное пространство второго (клонирование). Для большого адресного пространства эта операция занимает много времени. Кроме того, два процесса должны создать канал для обмена данными. Не во всех ОС это осуществляется быстро и легко.

Бывают ситуации, когда более выгоден другой подход к достижению параллельности, а именно, ***многопоточный процесс*** (***multithreaded process***). У многопоточного процесса имеется два и более потоков (счетчиков команд), которые совместно используют одно и то же адресное пространство и описатели объектов.

Каждый процесс NT создается с одним потоком. При необходимости можно создать внутри процесса дополнительные потоки. Они часто используются для выполнения в программе ***асинхронных операций*** (***asynchronous operations***), т.е. операций, которые могут иметь место в любое время, безотносительно к основному течению программы. Например, можно использовать поток для периодического сохранения редактируемого документа или опроса устройства ввода, например мыши или клавиатуры. Используя один поток для выполнения основной программы, и создав второй для опроса устройства ввода, система может по отдельности выполнять обе операции.

Чтобы достичь параллелизма с использованием потоков, программа создает два или несколько потоков в одном процессе. Многопоточный компилятор изображен на рисунке (*Рисунок 2.9*).

Рисунок 2.9 Многопоточный компилятор

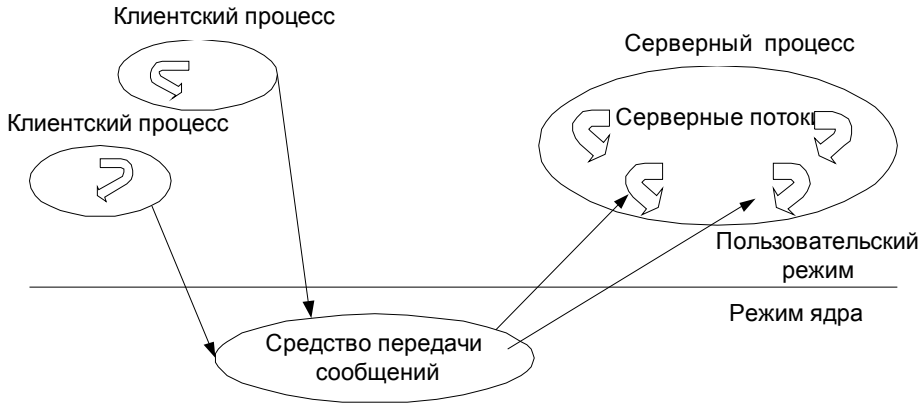


Многопоточные процессы добиваются параллельности и при этом не имеют недостатков, связанных с использованием двух процессов. Потоки требуют меньших издержек и создаются быстрее, чем процессы (их называют легковесными процессами). Кроме того, поскольку все потоки процесса используют одну и ту же память, за исключением своих стеков и содержимого регистров, не требуется никакого механизма обмена данными. Один поток просто записывает данные в память, а другой считывает их. Аналогично, все ресурсы процесса (объекты) одинаково доступны всем его потокам.

Для выбора порядка, в котором исполняются потоки, ядро NT использует приоритеты. Потоки с большим приоритетом исполняются прежде потоков с меньшим приоритетом. Ядро периодически изменяет приоритет каждого потока, чтобы гарантировать выполнение их всех.

Использование многопоточного процесса - идеальное решение для серверного приложения (например, защищенных подсистем Windows NT), которое принимает запросы от клиентов и выполняет по каждому запросу один и тот же код. Например, файл-сервер выполняет операции с файлами; он открывает файлы, читает, пишет в них и закрывает их. Хотя каждый запрос может требовать от сервера работы со своим файлом, программа сервера загружается в память только один раз. Каждый приходящий запрос принимается и обрабатывается отдельным потоком сервера. Все запросы клиентов обслуживаются параллельно (*Рисунок 2.10*).

Рисунок 2.10 Многопоточный сервер



Писать многопоточное приложение надо очень тщательно, так как все потоки процесса имеют полный доступ к его адресному пространству. Потоки могут пересечься друг с другом, выполняя чтение или запись в память не в том порядке. Если мы используем два процесса, которые взаимодействуют через канал, такого не произойдет. Поэтому защищенные подсистемы реализованы как отдельные процессы и соответственно у каждой из них отдельное адресное пространство и вмешательство других подсистем или пользовательских приложений не допускается.

Объект-поток

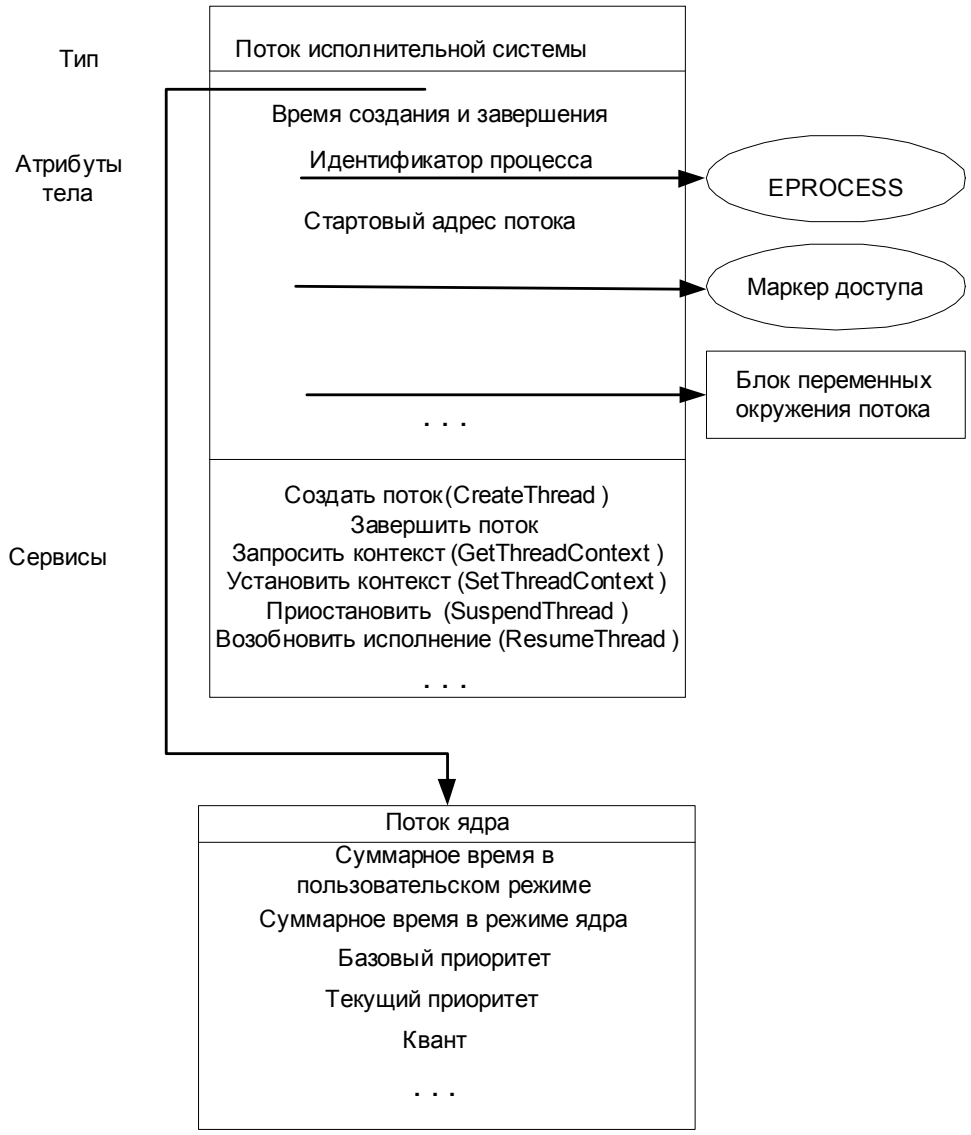
Процесс NT остается мертвым без потока, который можно направить на выполнение. Как и процессы, потоки исполнительной системы NT реализованы в виде объектов. Объект - поток изображен на рисунке (*Рисунок 2.11*). В исполнительной системе потоки - это объекты исполнительной системы, создаваемые и уничтожаемые диспетчером объектов. Диспетчер процессов определяет атрибуты, хранящиеся в

теле объектов потоков, а также предоставляет системные сервисы для чтения и изменения этих атрибутов. Основные атрибуты и сервисы для объектов

потоков приводятся на рисунке. Объект поток исполнительной системы включает объект поток ядра (содержит указатель на объект поток ядра). Ядро управляет объектом поток ядра, а исполнительная система управляет объектом исполнительной системы.

Основные атрибуты этих объектов приводятся по отдельности, для того чтобы лучше понять какие задачи решаются ядром, а какие исполнительной системой, так же как и для процесса.

Рисунок 2.11 Объект-поток

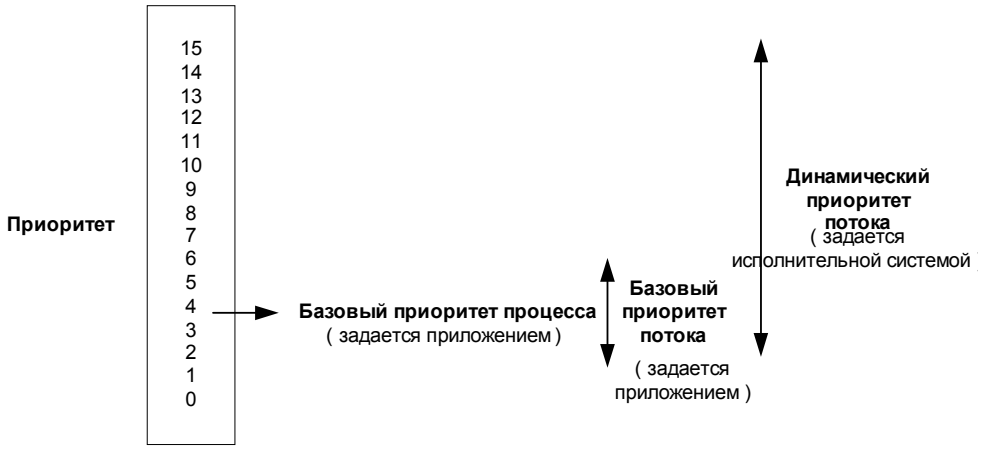


Рассмотрим основные атрибуты:

- *идентификатор клиента* - уникальное значение, идентифицирующее поток;
- *контекст потока* - набор значений регистров, определяющих состояние выполнения потока с учетом специфики конкретного процессора;
- *динамический приоритет* - приоритет планирования потока на данный момент;
- *базовый приоритет* - нижний придел динамического приоритета потока;
- *процессорное сродство потока* - набор процессоров, на которых может исполняться поток, подмножество процессорного сродства процесса потока;
- *время выполнения потока* - общее время выполнения потока в пользовательском режиме и режиме ядра.

Некоторые атрибуты потока напоминают атрибуты процесса. Одни атрибуты, такие как процессорное сродство потока и динамический приоритет, могут усилить или ослабить ограничения, задаваемые для процесса в целом. Например, процессорное сродство каждого потока - это подмножество процессорного сродства процесса. Аналогично, каждый поток имеет базовый приоритет, который лежит в диапазоне от двух уровней ниже базового приоритета процесса до двух уровней выше этого приоритета (*Рисунок 2.12*). На рисунке показан также динамический приоритет потока, нижней границей которого является базовый приоритет потока, а верхняя граница зависит от вида работ, исполняемых потоком.

Рисунок 2.12 Соотношение приоритетов



Планирование потоков

Теперь рассмотрим планирование коротко и более формально

В Windows реализовано вытесняющее планирование на основе уровней приоритета, при котором всегда выполняется поток с наибольшем приоритетом, готовый к выполнению. При выборе потока также учитывается привязка к процессорам.

Выбранный для выполнения поток работает в течение некоторого периода времени, называемого квантом. Квант определяет, сколько времени будет работать поток, пока ОС не прервет его. После этого ОС будет решать, искать другой готовый к выполнению поток с тем же приоритетом или снизить приоритет данного потока.

Поток может не полностью использовать свой квант. Если появляется поток с более высоким приоритетом готовый к выполнению, то текущий поток вытесняется, даже если не истек его квант.

Код, отвечающий за планирование, реализован в ядре. Набор процедур, которые этим занимаются, называются ***диспетчером ядра*** (***kernel dispatcher***). Диспетчеризация потоков инициализируется любым из следующих событий:

- поток готов к выполнению (он создан или вышел из состояния ожидания);
- поток вышел из состояния выполнения (истек его квант, он завершился или перешел в состояние ожидания);
- приоритет потока изменился;
- изменилась привязка к процессорам.

Квант

Квант - это интервал процессорного времени, отведенный потоку для выполнения. По истечении кванта ОС проверяет готов ли к выполнению другой поток с таким же приоритетом. Если такого нет, то ОС выделяет текущему потоку еще один квант.

Потокам выделяются разные кванты в зависимости от ОС (Windows 2000 Professional или Windows 2000 Server). По умолчанию начальная величина кванта в Windows 2000 Professional равна 6, а в Windows 2000 Server - 36.

Windows может динамически увеличивать величину кванта. Поясним это на примере. Например, вы начали пересчет большой таблицы и запустили игру. Если повысить приоритет игры, то фоновый процесс

пересчета будет получать очень маленькую часть процессорного времени. А увеличение кванта для игры не будет блокировать пересчет, просто игре будет отдано предпочтение.

Сценарии планирования

Самостоятельное переключение

Поток может самостоятельно освободить процессор, перейдя в состояние ожидания на каком-либо объекте (***WaitForSingleObject*** или ***WaitForMultipleObjects***).

Вытеснение

Поток с более низким приоритетом вытесняется готовым к волнению потоком с более высоким приоритетом. Такая ситуация может возникнуть в двух случаях:

- поток с более высоким приоритетом "дождался" (произошло событие, которого он ждал);
- приоритет потока увеличился или уменьшился.

Завершение кванта

Поток израсходовал свой квант процессорного времени, и ОС должна решить, следует ли понизить его приоритет и подключить к процессору другой поток.

Завершение потока

Завершаясь (после возврата из основной процедуры и вызова ***ExitThread*** или ***TerminateThread***) поток переходит в состояние Terminated.

Динамическое повышение приоритета

Динамическое повышение приоритета предназначено для оптимизации общей пропускной способности и отзывчивости системы, а также для устранения потенциально нечестных сценариев планирования. Но это не панацея и от него выигрывают не все приложения.

Windows может динамически повышать значение текущего приоритета потока в одном из следующих случаев:

- после завершения операции ввода/вывода;
- по окончании ожидания на каком-либо объекте исполнительной системы;
- при нехватке процессорного времени.

В первом случае ОС временно повышает приоритет потоков по окончании операций ввода/вывода. В этом случае у таких потоков появляется больше шансов возобновить выполнение и обработать полученные данные. После динамического повышения приоритета поток в течение одного кванта выполняется с этим приоритетом, после чего приоритет уменьшается на один уровень и потоку выделяется еще один квант. Этот цикл продолжается до тех пор пока приоритет не снизится до базового.

Во втором случае (***SetEvent***, ***ReleaseSemaphore***) приоритет потока увеличивается на один уровень.

В третьем случае рассмотрим следующий сценарий. Поток с приоритетом 7 не дает выполняться потоку с приоритетом 4, а при этом поток с приоритетом 11 ожидает ресурс, заблокированный потоком с приоритетом 4. В подобной ситуации диспетчер настройки баланса сканирует очереди готовых потов и ищет потоки, которые находились в состоянии Ready более 3 секунд. Обнаружив такой поток, диспетчер повышает его приоритет до 15 и выделяет ему квант вдвое больше обычного. По истечении двух квантов приоритет потока снижается до исходного уровня.

Абстрагирование приоритетов

Windows поддерживает шесть классов приоритета процесса: ***idle*** (***простаивающий***), ***below normal*** (***ниже обычного***), ***normal*** (***обычный***), ***above normal*** (***выше обычного***), ***high*** (***высокий***), ***real-time*** (***реального времени***).

Real-time - это наивысший возможный приоритет. Потоки в этом процессе обязаны немедленно реагировать на события. Исполнение таких потоков может привести к полной блокировке системы. Будьте очень осторожны с этим классом.

High - потоки в этом процессе тоже должны немедленно реагировать на события (этот класс присвоен Task Manager).

Above normal - класс приоритета промежуточный между normal и high (новый класс, введенный в Windows 2000).

Normal - потоки в этом процессе не предъявляют особых требований к выделению им процессорного времени.

Below normal - класс приоритета промежуточный между normal и idle (новый класс, введенный в Windows 2000).

Idle - потоки в этом процессе выполняются, когда система не занята другой работой. Этот класс приоритета обычно используется для утилит, работающих в фоновом режиме.

Кроме того, Windows поддерживает семь относительных приоритетов потоков: idle (простаивающий), lowest (низший), below normal (ниже обычного), normal (обычный), above normal (выше обычного), highest (высший) и time-critical (критичный по времени). Эти приоритеты относительны классу приоритета процесса. Относительные приоритеты потоков описаны в таблице (*Таблица 2.1*).

Таблица 2.1 Относительные приоритеты потоков

Класс приоритета процесса						
Относительный приоритет потока	Idle	Below normal	Normal	Above normal	High	Real-time
Time-critical	15	15	15	15	15	31
Highest	6	8	10	12	15	26
Above normal	5	7	9	11	14	25
Normal	4	6	8	10	13	24
Below normal	3	5	7	9	12	23
Lowest	2	4	6	8	11	22
Idle	1	1	1	1	1	16

ЛАБОРАТОРНАЯ РАБОТА 2

ПРОЦЕССЫ И ПОТОКИ (ПРОГРАММИРОВАНИЕ)

Теория

В этой лабораторной работе представлена методология создания ***параллельных приложений*** (***concurrent application***). Такое приложение может исполняться в двух и более местах одновременно. Написание параллельных приложений полезно по двум причинам. Во-первых, современное ПО разрабатывается по частям, которые могут выполняться параллельно. Во-вторых, когда приложение использует параллельную модель, современные ОС, такие как Windows NT, представляют много возможностей для разработчиков для управления параллельными приложениями. В этой лабораторной работе параллельность рассматривается на уровне процесса и потока. Обычные программы, которые вы писали, выполнялись как один процесс с одним потоком (последовательное программирование). Современное поколение ПО использует преимущества многих процессоров на вашем компьютере или доступных по сети. Таким образом, традиционные последовательные модели вычислений заменяются параллельными моделями со многими процессами и потоками. В лабораторной работе будет представлена информация о процессах и потоках Windows NT и рассмотрено создание процессов и потоков, используя Win32 API.

Создание процесса

Один процесс может создать другой, вызывая Win32 API CreateProcess (при этом вызове используются Native API NTCreateProcess и NTCreate Thread). Когда создается процесс ***исполнительная система*** (***Executive***) выполняет большой объем работы. Она выделяет новое адресное пространство и ресурсы для процесса, а также создает для него новый базовый поток. Когда новый процесс создан, старый процесс будет продолжать исполняться, используя старое адресное пространство, а новый будет выполняться в новом адресном пространстве с новым базовым потоком. Существует много различных опций для создания процесса, поэтому функция CreateProcess имеет много параметров, причем некоторые из них достаточно сложные. После того, как исполнительная система создала новый процесс, она возвращает его ***описатель*** (***handle***), а также описатель его базового потока.

Рассмотрим прототип функции **CreateProcess**. В прототипе не используются стандартные типы C, а вместо этого используется набор типов, определенный в ***windows.h***.

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,
// имя исполняемого модуля (указатель)
  LPTSTR lpCommandLine,
  // командная строка (указатель)
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  // атрибуты безопасности процесса (указатель)
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  // атрибуты безопасности потока (указатель)
  BOOL bInheritHandeles, // флаг наследования описателя
  DWORD dwCreationFlags,// флаги создания
  LPVOID lpEnvironment,
  // новый блок окружения (указатель)
  LPCTSTR lpCurrentDirectory,
  // имя текущей директории (указатель)
  LPSTARTUPINFO lpStartupInfo,
  // STARTUPINFO (указатель)
  LPPROCESS_INFORMATION lpProcessInformation
  // PROCESS_INFORMATION (указатель)
        )
```

Десять параметров **CreateProcess** обеспечивают большую гибкость в использовании для программиста, хотя в простейшем случае для многих параметров можно использовать значения ***по умолчанию*** (***by default***). Здесь мы рассмотрим относительно простой набор параметров, а более детально можно посмотреть в *Microsoft Developers Network* (*MSDN*).

Параметры lpApplicationName и lpCommandLine

Два первых параметра обеспечивают два различных способа определения имени файла, который будет выполняться базовым потоком процесса. **lpApplicationName** - эта строка содержит имя файла, который будет выполняться, а **lpCommandLine** - эта строка содержит командную строку для запуска процесса в cmd.exe. Существует набор правил, определяющий в каком случае какое имя использовать (см. MSDN). Будем задавать **NULL** для **lpApplicationName** и командную строку для **lpCommandLine**. Предположим, вы хотите создать процесс для запуска notepad.exe с файлом temp.txt.

```
#include <string.h>
strcpy(lpCommandLine,"C:\\WINNT\\SYSTEM32\\NOTEPAD.EXE temp.txt");
CreateProcess(NULL, lpCommandLine, …);
```

Параметры lpProcessAttributes, lpThreadAttributes и bInheritHandeles

В данном случае будем использовать значения по умолчанию для атрибутов безопасности процесса и потока - **NULL** и **FALSE** для флага наследования (атрибуты безопасности и флаг наследования будут рассмотрены далее).

```
CreateProcess(NULL, lpCommandLine, NULL, NULL, FALSE, …);
```

Параметр DwCreationFlags

Этот параметр используется для управления приоритетом нового процесса и другими особенностями.

Порожденный процесс - ***процесс-ребенок*** (***child***) может быть создан с одним из четырех классов приоритетов **HIGH_PRIORITY_CLASS**, **IDLE_PRIORITY_CLASS**, **NORMAL_PRIORITY_CLASS** или **REALTIME_PRIORITY_CLASS**. Значение по умолчанию - **NORMAL_PRIORITY_CLASS**, но если ***порождающий процесс*** (***процесс родитель parent***) имеет приоритет **IDLE_PRIORITY_CLASS**, то и процесс-ребенок также будет иметь приоритет **IDLE_PRIORITY_CLASS**.

Потоки процесса с приоритетом **HIGH_PRIORITY_CLASS** будут вытеснять потоки процессов с приоритетами **IDLE_PRIORITY_CLASS** или **NORMAL_PRIORITY_CLASS**. Потоки процесса с приоритетом **IDLE_PRIORITY_CLASS** будут выполняться только в том случае, если нет других потоков для выполнения.

Потоки процесса с приоритетом **REAL_PRIORITY_CLASS** будут вытеснять потоки всех других классов.

Можно задать и другие флаги, а также использовать вместе (все возможные значения см. в MSDN). Здесь нам будет полезен флаг **CREATE_NEW_CONSOLE** - новый процесс будет создан в своем собственном окне cmd.exe.

Для создания нового процесса (child) с высоким приоритетом в его собственном окне используйте - **HIGH_PRIORITY_CLASS | CREATE_NEW_CONSOLE**.

```
CreateProcess(NULL, lpCommandLine, NULL, NULL, FALSE,
HIGH_PRIORITY_CLASS |     CREATE_NEW_CONSOLE, …);
```

Параметр lpEnvironment

Используется для передачи нового блока переменных окружения порожденному процессу-ребенку (child). Если **NULL**, то ребенок использует тоже окружение, что и родитель. Если не **NULL**, то lpEnvironment должен указывать на массив строк, каждая **name=value**. Здесь мы будем использовать **NULL**.

```
CreateProcess(NULL, lpCommandLine, NULL, NULL, FALSE,
HIGH_PRIORITY_CLASS | CREATE_NEW_CONSOLE, NULL, …);
```

Параметр lpCurrentDirectory

Определяет полное путевое имя директории, в которой ребенок будет выполняться. Если использовать **NULL**, то ребенок будет использовать директорию родителя.

```
CreateProcess(NULL, lpCommandLine, NULL, NULL, FALSE,
HIGH_PRIORITY_CLASS | CREATE_NEW_CONSOLE, NULL, NULL, …);
Параметр lpStartupInfo
Это указатель на следующую структуру:
typedef struct_STARTUPINFO { //si
  DWORD cb; // длина структуры
  LPTSTR lpReserved;
  LPTSTR lpDesktop;
  LPTSTR lpTitle;
  DWORD dwX;
  DWORD dwY;
  DWORD dwXSize;
  DWORD dwSizeY;
  DWORD dwXCountChars;
  DWORD dwYCountChars;
  DWORD dwFillAttribute;
  DWORD dwFlags;
  DWORD dwShowWindow;
  WORD    cbReserved2;
  LPBYTE lpReserved2;
  HANDLE hStdInput;
  HANDLE hStdOutput;
  HANDLE hStdError; } STARTUPINFO, *LPSTARTUPINFO;
```

Экземпляр этой структуры данных должен быть создан в вызывающей программе (описание полей структуры см. в MSDN). Затем ее адрес должен быть передан как параметр в CreateProcess.

Внимание!!! CreateProcess не знает длины структуры, поэтому обязательно инициализируйте поле размера в **STARTUPINFO** (поле .cb) до того как передать указатель.

```
STARTUPINFO startupInfo;
…
ZeroMemory(&startupInfo,sizeof(STARTUPINFO));
startupInfo.cb=sizeof(startupInfo);
CreateProcess(NULL, lpCommandLine, NULL, NULL, FALSE,
 HIGH_PRIORITY_CLASS | CREATE_NEW_CONSOLE, NULL, NULL, &startupInfo, …);
```

Параметр lpProcessInformation

Это - указатель на структуру:

```
Typedef struct_PROCESS_INFORMATION {
   HANDLE hProcess;
   HANDLE hThread;
   DWORD  dwProcessId;
   DWORD  dwThreadId; } PROCESS_INFORMATION;
```

В этой структуре нет поля длины, поэтому процессу нужно только создать экземпляр **PROCESS_INFORMATION** и передать указатель на него функции CreateProcess. После возврата, поля структуры будут заполнены, и вы получите:

- Описатель вновь созданного процесса (**hProcess**)
- Описатель его базового потока (**hThread**)
- Глобальный идентификатор процесса (**dwProcessId**)
- Глобальный идентификатор потока (**dwThreadId**).

CreateProcess отводит место под объекты процесс и поток и возвращает значения их описателей (индексы в таблице) в структуре **PROCESS_INFORMATION**. Вы можете освободить выделенное место, используя **CloseHandle**. Конечно, когда процесс заканчивается, все его описатели автоматически освобождаются. Хотя описатель это системный ресурс, хорошая практика, если вы будете закрывать описатель, если не используете его.

CreateProcess возвращает ноль, если все плохо и не ноль в противном случае.

Далее приводится код для создания процесса:

```
#include <windows.h>
#include <stdio.h>
#include <string.h>
…
STARTUPINFO startInfo;
PROCESS_INFORMATION processInfo;
…
strcpy(lpCommandLine,"C:\\WINNT\\SYSTEM32\\NOTEPAD.EXE temp.txt");
ZeroMemory(&startupInfo,sizeof(STARTUPINFO));
startupInfo.cb=sizeof(startupInfo);
```

```
if(!CreateProcess(NULL, lpCommandLine, NULL, NULL, FALSE,
HIGH_PRIORITY_CLASS | CREATE_NEW_CONSOLE, NULL, NULL, &startupInfo,
&processInfo))
{
fprintf(stderr,"CreateProcess failed on error %d\n",GetLastError());
exitProcess(1);
}
…
CloseHandle(&processInfo.hThread);
CloseHandle(&processInfo.hProcess);
```

Создание потока

Вы можете создать дополнительные потоки в текущем процессе, используя функцию **CreateThread** Win32 API (которая использует *NTCreateThread* Native API). Каждый поток это отдельная исполняемая сущность внутри адресного пространства процесса. Для создания потока программист должен задать необходимую информацию.

Рассмотрим прототип функции:

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  // атрибуты безопасности потока (указатель)
  DWORD dwStackSize,
  // начальный размер стека потока в байтах
  LPTHREAD_START_ROUTINE lpStartAddress,
  // функция потока (указатель)
  LPVOID lpParameter, // аргумент для нового потока
  DWORD dwCreationFlags, // флаги создания
  LPDWORD lpThreadId
  // возвращаемый идентификатор потока (указатель)
       );
```

Этот прототип использует шесть параметров для описания характеристик нового потока. Функция создает описатель потока, и возвращать его как результат вызова. Это означает, что системные ресурсы размещаются при удачном вызове CreateThread и программист должен закрыть описатель, когда он больше не используется.

Параметр lpThreadAttributes

Это то же самое, что и атрибуты безопасности в CreateProcess только для потока. В простейшем случае NULL (будет рассмотрено далее).

```
CreateThread(NULL, …);
```

Параметр dwStackSize

Каждый поток выполняется независимо от других потоков процесса и имеет свой собственный стек. Используя этот параметр, программист может установить размер стека. Обычно используется значение по умолчанию - 0.

```
CreateThread(NULL, 0,  …);
```

Параметры lpStartAddress и lpParameter

Для процесса необходимо было задать имя выполняемого файла. Для потока необходимо задать адрес в текущем адресном пространстве, где новый поток должен выполняться. **LpStartAddress** - это и есть такой адрес. В языках программирования (таких как C) обычно невозможно задать в качестве стартового адреса адрес в середине процедуры. Нужно использовать ***точку входа в процедуру*** (***entry point***). **LpStartAddress** - это точка входа для функции, которая имеет следующий прототип:

```
DWORD WINAPI ThreadFunc(LPVOID);
```

Как передать параметр функции, которая будет выполняться новым потоком? Необходимо использовать **lpParameter**, он будет передан функции как параметр, когда новый поток начнет ее выполнение.

Пусть дан прототип функции, которую будет выполнять новый поток:

```
DWORD WINAPI myFunc(LPVOID);
```

Предположим, что родительский поток хочет передать целый аргумент новому детскому потоку:

```
int theArg;
…
CreateThread(NULL, 0, myFunc, &theArg, …);
```

Параметр dwCreationFlags

Этот параметр используется для определения, каким образом создать новый поток. Возможное значение CREATE_SUSPENDED - в этом случае новый поток будет создан, но приостановлен до тех пор, пока другой поток не выполнит вызов:

```
ResumeThread(targetThreadHandle);
```

Здесь **targetThreadHandle** - описатель нового потока. Значение по умолчанию 0 - в этом случае поток будет активизирован сразу после создания.

```
CreateThread(NULL, 0, myFunc, &theArg, 0, …);
```

Параметр lpThreadId

Это - указатель на идентификатор потока (аналог поля **dwThreadId** в структуре **PROCESS_INFORMATION**, которую возвращает **CreateProcess**).

```
DWORD targetThreadId;
…
CreateThread(NULL, 0, myFunc, &theArg, 0, & targetThreadId);
```

Чему нужно научиться?

Интерпретатор командной строки (cmd.exe в NT, command.com в DOS или один из shell UNIX) это программа, которая обеспечивает текстовый интерфейс для общения пользователя с ОС. Пользователь запускает программу из командной строки интерпретатора, печатая имя файла, который содержит выполняемую программу, с набором параметров. Это таже самая информация, которая задается параметром lpCommandLine в CreateProcess. Интерпретатор разбирает строку, чтобы получить имя файла, а затем заставляет процесс выполнить команду. В UNIX shell процесс командной строки действительно создает новый процесс, который загрузит и выполнит программу из файла. Интерпретатор командной строки ждет, пока команда выполнится, и ее поток закончится, перед тем как предоставить пользователю возможность дальше вводить команды.

Нужно научиться писать программы на C, используя функции CreateProcess и CreateThread для создания:

· одного или нескольких процессов (каждый с базовым потоком) - задание A;

· нескольких потоков в одном процессе - задание B.

Задания

Уровень 1 (A)

Используя функцию ***CreateProcess***, создайте процесс для запуска notepad.exe.

Уровень 2 (A)

Используйте текстовый редактор (например, notepad) для подготовки конфигурационного файла. Он может содержать следующий перечень:

```
C:\\WINNT\\SYSTEM32\\NOTEPAD.EXE our.txt
C:\\WINNT\\SYSTEM32\\CALC.EXE
C:\\WINNT\\SYSTEM32\\CHARMAP.EXE
```

Напишите программу, которая получает имя конфигурационного файла из командной строки, должна открыть конфигурационный файл, прочитать строки и создать процесс для запуска каждой команды.

Уровень 3 (А)

Необходимо выполнить задание Уровень 3 (В).

Уровень 1 (В)

Важно!!! Нужно использовать /MTd вместо /MLd, чтобы указать компилятору, что код будет многопоточным.

Создайте два дополнительных потока в вашем процессе. Пусть функция вашего первого потока в бесконечном цикле выводит сообщение (например, "Я поток 1 и я выполняюсь!"), а функция вашего второго потока в бесконечном цикле выводит сообщение (например, "Я поток 2 и я выполняюсь!"). Базовый поток после создания этих двух дополнительных потоков пусть тоже в бесконечном цикле выводит сообщение (например, "Я базовый поток и я выполняюсь!"). Добейтесь того, чтобы каждый из трех потоков поочередности выводил на экран эти слова.

Затем измените программу так, чтобы после создания двух дополнительных потоков, базовый поток заканчивал работу. Объясните полученный вывод.

Теперь измените программу так, чтобы после создания двух дополнительных потоков, базовый поток вызывал функцию **`Sleep`**. Объясните полученный вывод.

Уровень 2 (В)

Создать в цикле несколько дополнительных потоков в процессе. В качестве аргументов командной строки задать число создаваемых потоков и время их существования. Пусть функция потока в бесконечном цикле выводит сообщение (например, "Я поток с номером N и я выполняюсь!").

Используйте системное время. Время, которое процесс и потоки в нем должны существовать, задается как параметр, и по истечении этого времени сообщество должно погибнуть. После того как рабочие потоки созданы и выполняют свою работу, поток координатор проверяет текущее время, чтобы определить, не наступил ли назначенный час. Если нет, то он засыпает на 1сек, а потом просыпается и проверяет время снова. Когда время истечет поток координатор устанавливает

глобальную переменную **runFlag=FALSE**, ждет 5сек и заканчивается. Используйте разделяемую всеми потоками глобальную переменную **runFlag**.

Уровень 3 (В)

Разработайте программу, которая позволит изменять класс приоритета процесса и приоритеты потов этого процесса. Как значение приоритета влияет на выделение процессорного времени? Что будет, если запретить динамическое изменение приоритета?

Изучите и используйте в своей программе функции: Изучите и используйте в своей программе функции: **GetPriorityClass**, **SetPriorityClass**, **SetProcessPriorityBoost**, **GetProcessPriorityBoost**, **ExitProcess**, **TerminateProcess**, **GetThreadPriority**, **SetThreadPriority**, **SetThreadPriorityBoost**, **ThreadPriorityBoost**, **SuspendThread** и **ResumeThread**.

Глава

3

Объекты: Внутренняя организация

В конце 80-х объекты стали рекламироваться в качестве панацеи от всех проблем в программировании. Однако впервые они появились в конце 60-х в языках программирования, предназначенных для моделирования. Объектно-ориентированное прграммирование, которое обеспечивает способ представления и манипулирования как физическими, так и абстрактными объектами, является естественным подходом в этой области.

Операционные системы также работают с объектами. Их объектами являются аппаратные ресурсы (устройства ввода-вывода, память) или программные ресурсы (файлы, процессы, семафоры). При этом можно работать с каждым типом ресурсов по-своему, а можно сосредоточить управление ресурсами в одном месте, используя их сходство.

Рассмотрим типы объектов *исполнительной системы NT*, структуру объектов и то, как ими управляет диспетчер объектов, а также защиту объектов.

Объекты исполнительной системы

В *исполнительной системе NT* объект - это отдельный экземпляр статически определенного типа объектов, существующий во время выполнения. ***Тип объекта*** (***object type***), также называемый ***классом объектов*** (***object class***), включает набор атрибутов объекта и сервисы, работающие с образцами этого типа. ***Атрибут объекта*** (***object attribute***) - это поле данных внутри объекта, определяющее его состояние. ***Объектные сервисы*** (***object services***) - это способы манипулирования объектами, с их помощью считывают или изменяют

атрибуты объекта. Главное различие между объектом и структурой данных состоит в том, что внутренняя структура объекта скрыта и для извлечения данных из объекта или помещения в него информации необходимо использовать объектные сервисы. Вы не можете непосредственно считывать или изменять внутренние данные объекта. Таким образом, внутренняя реализация объекта отделяется от кода, который лишь использует данный объект.

При проектировании исполнительной системы Windows NT было принято решение использовать объекты дла представления системных ресурсов, потомучто они позволяют централизованно выполнить три важные задачи ОС:

- Присвоение системным ресурсам имен;
- Совместное использование системных ресурсов различными процессами;
- Защита системных ресурсов от несанкционированного доступа.

Не все структуры данных в исполнительной системе являются объектами, а только те, которые должны быть совместно используемыми, защищенными, именованными (видимыми) программам пользовательского режима. Структуры, используемые, например, одним компонентом исполнительной системы для реализации внутренних функций, не являются объектами.

Несмотря на то, что объекты используются для представления совместно используемых системных ресурсов, Windows NT не является объектно-ориентированной системой. Большая часть кода написана на C для обеспечения переносимости, а C непосредственно не поддерживает объектно-ориентированные конструкции.

Диспетчер объектов (**object manager**) - это компонент исполнительной системы NT, отвечающий за создание, удаление и защиту объектов. Диспетчер объектов централизует операции управления ресурсами и выполняет следующие задачи:

- Обеспечивает общий унифицированный механизм использования системных ресурсов;
- Сосредотачивает защиту объектов в одном месте (C2);
- Позволяет назначить цену использования объектов процессами;
- Задает унифицированные правила удержания объектов (т.е. объект продолжает существовать до тех пор, пока его использует хоть один процесс);

- Поддерживает требования разнообразных сред ОС.

В NT используются два типа объектов: ***объекты исполнительной системы*** (***executive object***) и ***объекты ядра*** (***kernel object***). Объекты исполнительной системы представляются различными компонентами исполнительной системы. Они доступны программам пользовательского режима. Объекты ядра - это более примитивный набор объектов, реализованный ядром. Эти объекты невидимы коду пользовательского режима, а создаются и используются только внутри исполнительной системы. Объекты ядра обеспечивают фундаментальные функции, которые могут выполняться только самым низким уровнем ОС - ядром. Рассмотрим объекты видимые в пользовательском режиме.

Таблица 3.1 Объекты исполнительной системы

Тип объекта	Реализующий компонент	Что представляет собой
Процесс (Process)	Диспетчер процессов	Вызов программы, включая адресное пространство и ресурсы для ее выполнения
Поток (Thread)	Диспетчер процессов	Исполняемая сущность внутри процесса
Секция (Section)	Диспетчер памяти	Область совместно используемой памяти
Файл (File)	Диспетчер ввода/вывода	Образец открытого файла
Маркер доступа (Access token)	Система защиты	Информация о правах доступа зарегистрировавшегося в системе пользователя
Событие (Event)	Вспомогательные сервисы исполнительной системы	Объявление о том, что произошло событие
Мутант	То же	Механизм обеспечения взаимного исключения
Семафор	То же	Счетчик, регулирующий число потоков, которые могут использовать некоторый ресурс
Таймер	То же	Счетчик времени
Каталог объектов	Диспетчер объектов	Хранилище в памяти для имен объектов

Файловая модель

Обычно *объекты исполнительной системы* создаются либо защищенной подсистемой, в качестве непосредственной реакции на некоторое действие пользователя, либо различными компонентами ОС в процессе работы. Например, для того чтобы создать файл, приложение *Win32* вызывает функцию *Win32* API *CreateFile*(). В свою

очередь подсистема Win32 вызывает базовый сервис NT, создающий файловый объект исполнительной системы. Когда затем приложение читает из файла или записывает в него, подсистема Win32 и исполнительная система используют объект файл для работы с файлом.

Работа с файлами - это нетипичный случай объекта NT, так как файлы являются постоянными ресурсами и расположены не в памяти. Однако модель работы с файлами, используемая в большинстве языков программирования, удобна для работы с объектами. Рассмотрим модель работы с файлами:

- Необходимо открыть файл, прежде чем можно будет выполнить его чтение или запись. Можно открыть существующий файл, либо создать новый, с указанным именем (можно указать полное путевое имя);
- При открытии файла, задается тип операций, которые будут с ним совершатся (чтение, запись, добавление к концу файла и т.д.);
- Файловая система открывает файл и возвращает его описатель. В последующих операциях используется этот описатель, а когда работа с файлом закончена его необходимо закрыть;
- Две программы могут совместно использовать файл, если они обе получили его описатель.

Объектная модель имитирует файловаю модель. Описатели называются ***описателями объектов*** (***object handles***), а сами объекты хранятся в памяти.

Объектная модель

Как и в большинстве операционных систем, единицей работы в NT является процесс. Каждому процессу выделяется набор ресурсов, позволяющий ему выполнять свою работу: поток, чтобы можно было выполнять программу, и адресное пространство для хранения кода и данных. В процессе работы поток может запросить для своего процесса дополнительные ресурсы путем создания объектов или открывая описатели существующих объектов. Описатели объектов уникальны для процесса и указывают на его доступ к системным ресурсам.

Подсистема *Win*32 - это процесс NT, который выступает как сервер для приложений Win32. Когда приложение вызывает функцию API *Win*32, которая создает объект, подсистема *Win*32 обращается к объектному сервису NT (*Диспетчер объектов - Object Manager*). И *диспетчер объектов* выполняет следующие функции:

· Выделяет память для объекта;

· Присоединяет к объекту ***дескриптор защиты*** (***security descriptor***), который определяет, кому и как разрешено использовать объект;

· Создает и поддерживает ***каталог объектов***, где хранятся имена объектов;

· Создает ***описатель объекта*** и возвращает его приложению.

Все процессы пользовательского режима, включая подсистемы среды, должны получить описатель объекта, прежде чем их потоки смогут использовать этот объект. Описатели служат косвенными указателями на системные ресурсы; эта косвенность предотвращает непосредственный доступ приложений к системным структурам данных. Кроме того, описатели объектов создают дополнительные преимущест:

- Нет никаких различий между описателем файла, события или процесса за исключением того, на что они ссылаются. Можно работать с различными типами объектов однотипно;

- Только диспетчер объектов создает описатели и производит поиск объекта по его описателю.

Таким образом, любое действие в пользовательском режиме, затрагивающее объект, контролируется диспетчером объектов. Диспетчер объектов:

· Защищает объекты. Всякий раз, когда поток использует описатель объекта, диспетчер объектов проверяет, есть ли у потока право использовать объект, так как он хочет;

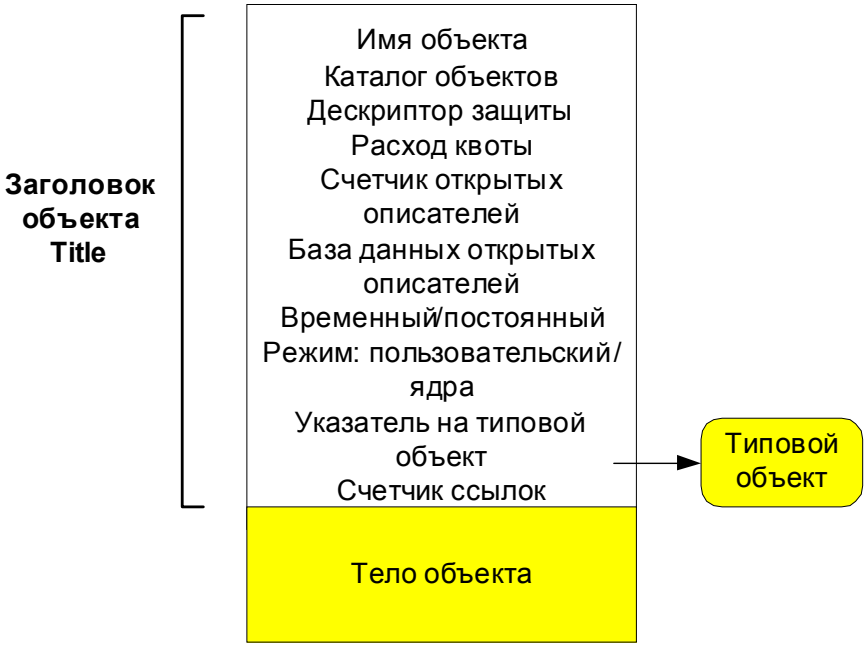
· Контролирует, кто использует объект. Не используемые объекты удаляются, но если у какого-либо процесса имеется описатель этого объекта (или у системы есть указатель на него), то он не будет удален;

· Контролирует использование ресурсов. Каждый раз, когда поток открывает описатель объекта, диспетчер объектов списывает с процесса этого потока объем памяти, используемый объектом. Объем использования памяти потоками процесса не может превышать ограничений памяти - ***квот*** (***quotas***), назначенных системным администратором пользователю, который представлен данным процессом.

Структура объектов

Каждый объект NT принадлежит к определенному типу объектов. Тип объекта определяет, какие данные содержит объект, а также какие сервисы могут к нему применяться. Для универсальности обработки разных объектов диспетчеру объектов необходимо, чтобы каждый объект содержал в заданном месте несколько полей со стандартной информацией. Поэтому объект разделен на две части - заголовок и тело (*Рисунок 3.1*). Диспетчер объектов работает с заголовком объекта, а другие компоненты *исполнительной системы* - с телами объектов создаваемых ими типов.

Рисунок 3.1 Содержимое заголовка объекта



Имя объекта делает его видимым другим процессам для совместного использования.

Каталог объектов обеспечивает иерархическую структуру, в которой хранятся имена объектов. *Дескриптор защиты* определяет, кто и каким образом может использовать данный объект.

Расход квоты задает квоту на использование ресурсов, которая списывается с процесса, когда тото открывает описатель данного объекта.

Счетчик открытых описателей подсчитывает количество открытых описателей данного объекта.

База данных открытых описателей содержит список процессов, открывших описатели данного объекта.

Временный/ постоянный статус указывает можно ли уничтожить имя и освободить память объекта, если он больше не используется.

Режим пользовательский/ядра определяет, в каком режиме доступен объект. *Указатель на типовой объект* ссылается на типовой объект, который содержит атрибуты, общие для набора однотипных объектов.

Счетчик ссылок - подсчитывает, сколько раз компоненты режима ядра ссылались на адрес данного объекта.

Диспетчер объектов предоставляет набор сервисов общего назначения, которые работают с атрибутами заголовка объекта. Их можно использовать с объектами любых типов. Часть этих универсальных сервисов подсистема *Win*32 предоставляет пользовательским приложениям *Win* 32.

Универсальные объектные сервисы:

- **З***акрыть* - закрывает описатель объекта;
- *Дублировать* - обеспечивает совместное использование объекта путем дублирования его описателя и передачи его копии другому процессу;
- *Опросить объект* - получить информацию о стандартных атрибутах объекта.
- *Опросить защиту* - получает дескриптор защиты;
- Установить защиту - *изменяет параметры защиты объекта*;
- *Ждать одного объекта* - синхронизирует выполнение потока с одним объектом;
- *Ждать несколько объектов* - синхронизирует выполнение потока с несколькими объектами.

Кроме заголовка каждый объект имеет тело, формат и содержимое которого определяется типом объекта. Тела всех объектов одного типа имеют одинаковый формат. Например, *диспетчер процессов* определяет тело объекта-процесса и обеспечивает сервисы для работы с хранящимися в нем данными.

Типовой объект

В заголовке объекта хранятся данные, формат которых одинаков для всех объектов, а значения могут быть различными. Например, у каждого объекта есть уникальное имя и может быть уникальный дескриптор защиты. Однако, есть данные, которые постоянны для всех объектов данного типа.Например, при открытии описателя объекта данного типа можно выбирать права доступа из некоторого набора прав, специфичных для этого типа. *Исполнительная система NT* поддерживает права доступа, "завершить" и "приостановить"- для объектов-потоков, и права доступа: "чтение", "запись" и "удаление", для файловых объектов.

В целях экономии памяти *диспетчер объектов* задает эти статические, типозависимые атрибуты один раз при создании нового типа объектов. Для хранения этих данных он использует специальный ***типовой объект*** (***type object***). Кроме того, *типовой объект* связывает друг с другом все объекты одного типа и при необходимости *диспетчер объектов* может перебрать их все.

Атрибуты тела типового объекта:

· ***Имя типа объекта*** - название объектов данного типа ("процесс", "событие" и т.д.);

· ***Типы доступа*** - типы доступа, которые могут быть запрошены потоком при открытии описателя объекта данного типа ("чтение", "запись", "завершить", "приостановить" и т.д.);

· ***Возможности синхронизации*** - может ли поток ожидать у объектов данного типа;

· ***Резидентный/ нерезидентный*** - могут ли объекты данного типа выгружаться из памяти.

Некоторые из этих атрибутов видимы посредством базовых сервисов и функций API Win32. *Синхронизация*, один из таких атрибутов и он указывает на то, что поток может синхронизировать свое выполнение, ожидая пока не изменится состояние объекта. Поток синхронизируется со следующими объектами *исполнительной системы*: процесс поток событие и т.д.

Итак, объекты *исполнительной системы NT* состоят из двух частей: заголовка объекта, управляемого *диспетчером объектов*, и тела объекта, которое управляется компонентом *исполнительной системы*,

создавшим данный тип. Одним из атрибутов заголовка объекта является указатель на типовой объект - структуру, определяющую статические атрибуты объектов данного типа.

Управление объектами

Диспетчер объектов предоставляет набор универсальных сервисов, применимых к объектам любого типа. Кроме того, другие компоненты *исполнительной системы NT* обеспечивают типозависимые сервисы для создаваемых ими типов объектов. Эти сервисы вызывают *диспетчер объектов* посредством внутренних интерфейсов. Следовательно, все сервисы, которые работают с объектами, должны пройти через *диспетчер объектов*.

Имена объектов

При большом количестве объектов необходима эффективная система их отслеживания. Для этой цели *диспетчеру объектов* нужны:

- Способ отличить один объект от другого;
- Метод поиска и выбора заданного объекта.

Первое требование выполняется благодаря тому, что объектам можно назначать имена. Большинство ОС позволяют именовать только файлы, каналы или блоки совместно используемой памяти. В отличие от них, *исполнительная система NT* позволяет назначить имя любому объекту.

Второе требование, возможность поиска объекта, также выполняется благодаря тому, что объекты имеют имена. Так как диспетчер объектов хранит объекты по именам, то он может и провести поиск по имени.

Имена объектов дают еще и возможность процессам совместно использовать объекты. Пространство имен объектов *исполнительной системы NT* является глобальным, доступным всем процессам в системе. Один процесс может создать объект и поместить его имя в глобальное пространство имен, а другой процесс может открытьописатель этого объекта, указав его имя. Если нет необходимости использовать объект совместно, то при создании имя указывате не нужно.

Диспетчер объектов не ищет имя объекта всякий раз, когда кто-нибудь использует объект. Поиск по имени осуществляется лишь в двух случаях:

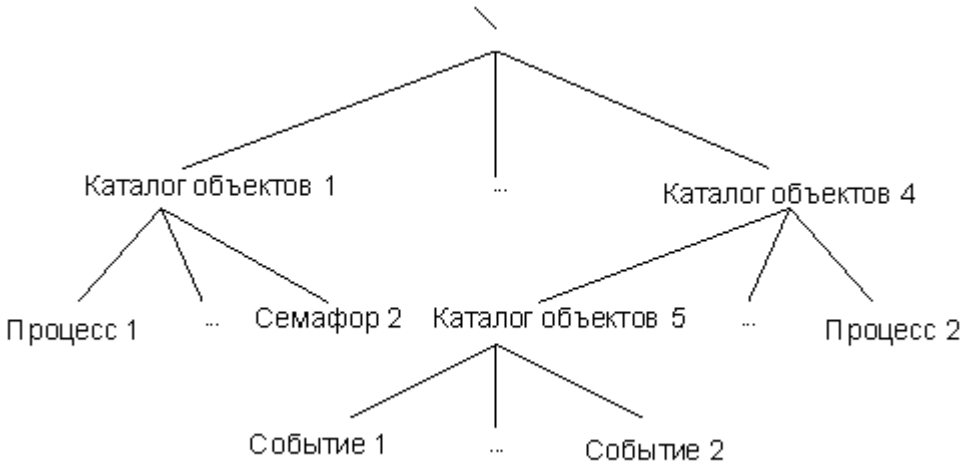
· Во-первых, когда процесс создает новый объект, *диспетчер объектов,* прежде чем поместить имя в глобальное пространство имен, осуществляет поиск по нему, чтобы убедиться, что оно уже не присвоено другому объекту;

· Во-вторых, когда процесс открывает описатель именованного объекта, *диспетчер объектов* осуществляет поиск по имени, находит объект и возвращает его описатель; после этого для ссылок на объект вызывающий процесс использует возвращенный описатель.

Имена объектов являются глобальными для данного компьютера, но не видимы в сети.

Объект - каталог объектов (***object directory object***) - это средство поддержки иерархической структуры имен *диспетчером объектов*. Он является аналогом каталога файловой системы и содержит имена других объектов, возможно и других каталогов объектов (*Рисунок 3.2*).

Рисунок 3.2 Иерархия имен объектов

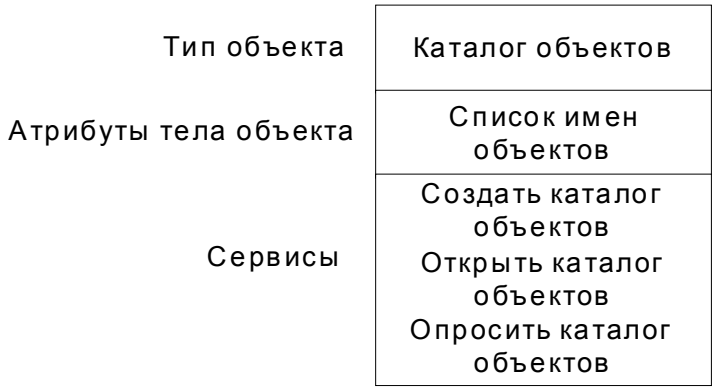


Сервисы создания и открытия используются для создания каталогов объектов и открытия их описателей. После того, как поток открыл описатель каталога объектов (с доступом по записи), он можем создавать другие объекты и помещать их в этот каталог.

Сервис опроса позволяет просматривать список объектов, хранящихся в каталоге. Объект-каталог объектов содержит информацию для трансляции имен объектов в указатели на них (*Рисунок 3.3*). Диспетчер объектов использует эти указатели для создания описателей объектов, которые он возвращает программам пользовательского режима.

В NT сервисы создания, открытия и опроса реализованы отдельно для каждого типа объектов.

Рисунок 3.3 Объект каталог объектов



Описатели объектов

Имена объектов важны для хранения и совместного использования объектов, но используются они не часто. Процесс указывает имя объекта, когда он создает объект или открывает его описатель. После этого процесс использует описатель объекта. Ссылка на объект при помощи описателя выполняется быстрее, чем по имени, так как диспетчер объектов не производит поиск имени.

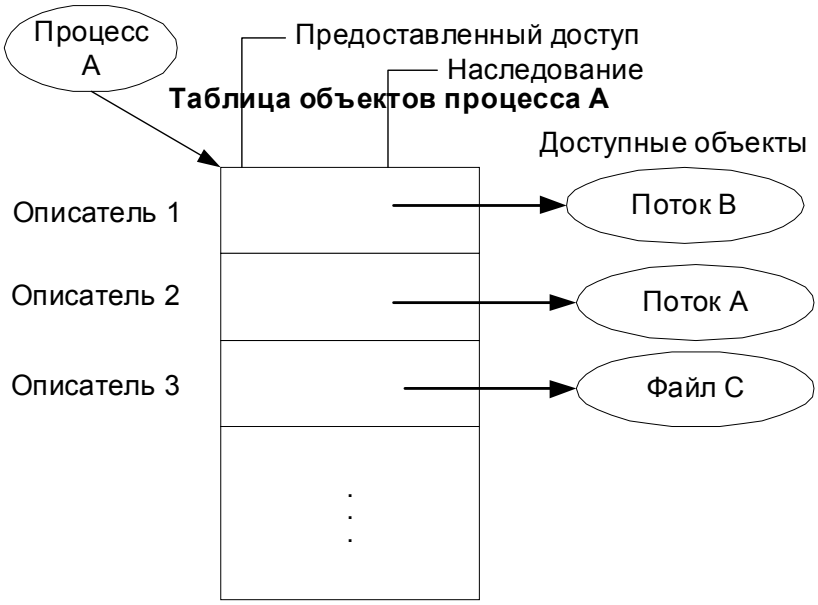
Описатель объекта NT - это индекс в *таблице объектов* (*object table*)процесса. Таблица объектов процесса содержит указатели на все объекты, описатели которых открыты процессом (*Рисунок 3.4*). Процесс может получить описатель объекта:

- создав объект;
- открыв описатель существующего объекта;
- унаследовав описатель от другого процесса;
- получив дубликат описателя из другого процесса.

Каждый вход таблицы процессов содержит предоставленные права доступа для соответствующего описателя и его ***режим наследования*** (***inheritance designation***). Режим наследования, определяет, получат ли процессы, созданные данным процессом, копию этого описателя в своих таблицах объектов. Хотя термин ***описатель*** (***handle***) означает индекс, его используют и для обозначения данных, хранящихся в соответствующем входе таблицы.

Два процесса совместно используют объект, если они оба открыли его описатель. Каждый из этих описателей уникален.

Рисунок 3.4 Структура таблицы объектов



Удержание объектов

Так как все процессы пользовательского режима, осуществляющие доступ к некоторому объекту, должны вначале открыть его описатель, то диспетчер объектов может отслеживать, сколько процессов, и какие именно, используют данный объект. Отслеживание открытых описателей - это первый шаг в реализации ***удержания объекта*** (***object retention***) , т.е. сохранения объектов только на то время, пока они используются, с последующим удалением.

Удержание объектов включает две фазы. Первая фаза называется удержанием имени (name retention). Всякий раз, когда процесс открывает описатель объекта, диспетчер объектов увеличивает счетчик открытых описателей в заголовке объекта. После того, как процесс закончил работу с объектом и закрыл имеющийся у него описатель объекта, диспетчер объектов уменьшает счетчик открытых описателей в заголовке объекта. Когда счетчик обнуляется, диспетчер объектов удаляет объект из своего пространства имен.

Вторая фаза удержания объектов - это прекращение удержания (т.е. удаление объектов), когда они больше не используются. Так как ОС осуществляет доступ к объектам посредством указателей, а не описателей, диспетчер объектов должен учитывать количество указателей на объект, которые он передал процессам ОС. Всякий раз

при выдаче указателя на объект диспетчер объектов увеличивает счетчик ссылок (reference count). Когда поток ОС заканчивает работу с объектом, он обращается к диспетчеру объектов для уменьшения счетчика ссылок. Таким образом, даже после того, как счетчик описателей достиг нуля, счетчик ссылок может оставаться положительным, указывая, что ОС продолжает использовать объект. Когда счетчик ссылок обнуляется, диспетчер объектов удаляет объект из памяти.

Программистам, разрабатывающим приложения, которые состоят из двух или более взаимодействующих между собой процессов, не нужно беспокоиться о том, что один процесс может удалить объект, прежде чем другой закончит работу с ним. Кроме того, закрытие описателей объекта приложениями не приводит к удалению объекта, если его продолжает использовать ОС. Например, пусть один процесс создал другой процесс для выполнения некоторой программы в фоновом режиме. Сразу же после этого первый процесс закрывает описатель второго. Так как второй процесс нужен ОС для выполнения программы, то она сохраняет ссылку на объект процесс. Только после того, как фоновая программа закончит выполнение, диспетчер объектов уменьшит счетчик ссылок второго процесса и удалит этот процесс.

Учет использования ресурсов

Учет использования ресурсов, так же как и удержание объектов, связан с использованием описателей объектов. Если у объекта есть положительный счетчик открытых описателей, это означает, что некоторый процесс использует данный объект. Это также означает, что некоторый процесс "платит" за память, занятую объектом.

Многие ОС используют квоты для ограничения доступа процессов к системным ресурсам. Диспетчер объектов NT централизованный учет использования ресурсов. Каждому пользователю назначается предельный размер квоты - ограничение на объем памяти, который могут использовать его процессы. Заголовок каждого объекта содержит атрибут расход квоты. Это значение диспетчер объектов вычитает из выделенной процессу квоты, когда поток этого процесса открывает описатель этого объекта. Если процессы пользователя открыли слишком много описателей и израсходовали всю его квоту, то некоторые описатели необходимо закрыть, прежде чем удастся открыть новые.

Защита объектов

Операционная сиситема должна быть защищена от нападения сразу на нескольких фронтах. Многопользовательская система должна защищать память, файла и другие ресурсы одного пользователя от других пользователей. Она должна защищать собственные данные, файлы и память от пользовательских программ. Кроме того, она должна отслеживать попытки обхода защиты. Министерство оборона США определило семь уровней защиты для ОС. Для соответствия уровню C2 в Windows NT должны быть следующие средства:

- **Защищенная регистрация в системе.** Необходимо, чтобы пользователь идентифицировал себя, введя уникальное имя и пароль, прежде чем ему будет предоставлен доступ к системе;
- **Селективный контроль доступа**. Владелец ресурса определяет, кто имеет доступ к данному ресурсу и задает тип доступа, назначая права доступа пользователю или группе пользователей;
- **Аудит.** Существует возможность обнаружения и регистрации важныж событий, имеющих отношение к защите, или любой попытки создания, использования или удаления системных ресурсов. Учет пользователей, выполнивших зарегистрированное действие, шпроизводится используя их идентификаторы;
- **Защита памяти.** Предотвращается чтение информации, записанной кем-либо в память, после того как этот блок памяти был возвращен ОС. Перед повторным использованием память реинициализируется

Суть селективного контроля доступа и аудита состоит в защите объектов. Главная идея системы защиты Windows NT - это создание шлюза, через который должен пройти каждый пользователь системных ресурсов. Так как все системные ресурсы, защита которых может быть нарушена, реализованы как объекты, то таким шлюзом является диспетчер объектов.

Рассмотрим защиту объектов с двух точек зрения:

- Идентификация пользователей;
- Управление доступом пользователей к объектам.

Маркеры доступа

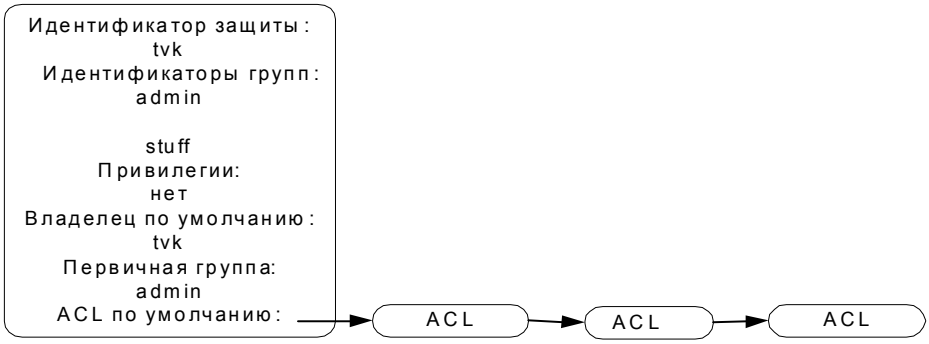
Первая линия защиты Windows NT - это процесс регистрации. ***Подсистема защиты*** (***security subsystem***) отвечает за ***аутентификацию*** (***authenticating***) пользователей, т.е. за проверку

того, что введенная пользователем информация при регистрации совпадает с информацией, хранящейся в базе данных защиты. После того, как подсистема защиты определила, что регистрация прошла успешно, она создает объект, который остается постоянно связанным с пользовательским процессом. Этот объект называется ***маркером доступа*** (***access token***). Маркер доступа служит официальным удостоверением личности процесса, когда тот пытается использовать какой-либо системный ресурс. Он идентифицирует процесс и его потоки для ОС. Пример маркера доступа показан на рисунке (*Рисунок 3.5*).

Каждый вход таблицы процессов содержит предоставленные права доступа для соответствующего описателя и его ***режим наследования*** (***inheritance designation***). Режим наследования, определяет, получат ли процессы, созданные данным процессом, копию этого описателя в своих таблицах объектов. Хотя термин ***описатель*** (***handle***) означает индекс, его используют и для обозначения данных, хранящихся в соответствующем входе таблицы.

Два процесса совместно используют объект, если они оба открыли его описатель. Каждый из этих описателей уникален.

Рисунок 3.5 Пример маркера доступа



Первый атрибут - это личный пользовательский ***идентификатор защиты*** (***security ID***), соответствующий идентификатору, указываемому пользователем при регистрации. Второй атрибут - это список групп, к которым принадлежит пользователь tvk.

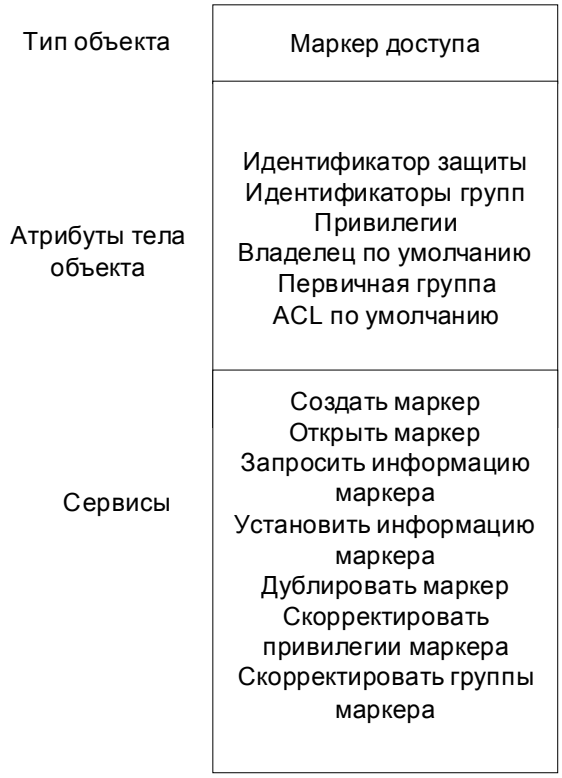
При попытке процесса открыть описатель объекта диспетчер объектов вызывает справочный монитор защиты. Справочный монитор защиты получает маркер доступа, связанный с прцессом, и использует его идентификатор защиты и список групп, чтобы определить имеет ли процесс право доступа к объекту.

Небольшое количество чувствительных к защите системных сервисов (например, создание маркера), также защищены от использования. Атрибут привилегий перечисляет все такие сервисы, к которым имеет право обращаться пользователь. Большинство пользователей привилегий не имеют.

Пользователь, создавший объект, становится его владельцем и может решать, кто еще имеет доступ к объекту. ***Список контроля доступа*** (***access control list***), по умолчанию хранящийся в маркере доступа - это первоначальный список прав доступа, который присоединяется к создаваемым пользователем объектам.

Рассмотрим атрибуты и сервисы объекта маркер доступа (*Рисунок 3.6*).

Рисунок 3.6 Объект маркер доступа



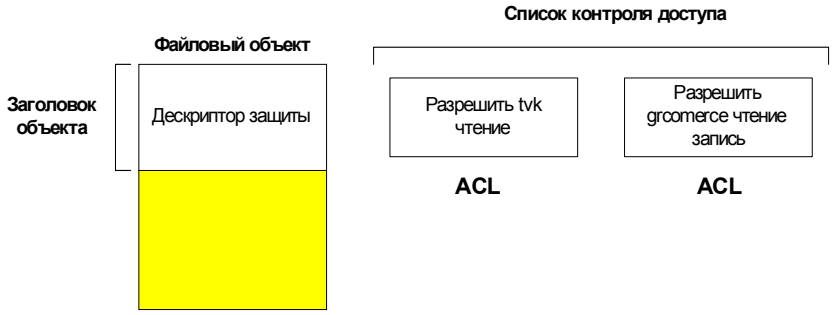
Списки контроля доступа

При создании любого объекта ему присваивается дескриптор защиты. Основной частью дескриптора защиты является ***список контроля доступа*** (***ACL***). Владелец объекта, который обычно является его создателем, обладает правом селективного контроля доступа к объекту и может изменять ACL объекта, чтобы разрешить или запретить другим его использование.

Каждые вход ACL называется ***элементом контроля доступа*** (***access control entry***). ACE содержит идентификатор защиты и набор прав доступа (*Рисунок 3.7*). Пользователю с соответствующим идентификатором защиты перечисленные права могут быть разрешены, запрещены или разрешены с аудитом. Сумма прав доступа, предоставленых отдельными ACE, формирует набор прав доступа, предоставляемых ACL.

Предположим, что вы хотите просмотреть файл. Если ACL файла содержит ACE с вашим идентификатором или с идентификатором одной из ваших групп, и этот ACE содержит право доступа "чтение", то просмотр файла вам разрешен.

Рисунок 3.7 Список контроля доступа



Система защиты назначает ACL новому объекту, применяя три взаимоисключающих правила, в приведенном ниже порядке:

- Если ACL явно задан при создании объекта, то система защиты присваивает объекту данный ACL;
- Если ACL не задан, а у объекта есть имя, то система защиты ищет ACL каталога объектов, в котором будет сохранено имя нового объекта, определяет те ACE, которые заданы наследуемыми и из них составляет ACL, который получит новый объект;
- Если не задано ни то ни другое, то система защиты присваивает новому объекту ACL по умолчанию из маркера доступа вызывающего процесса.

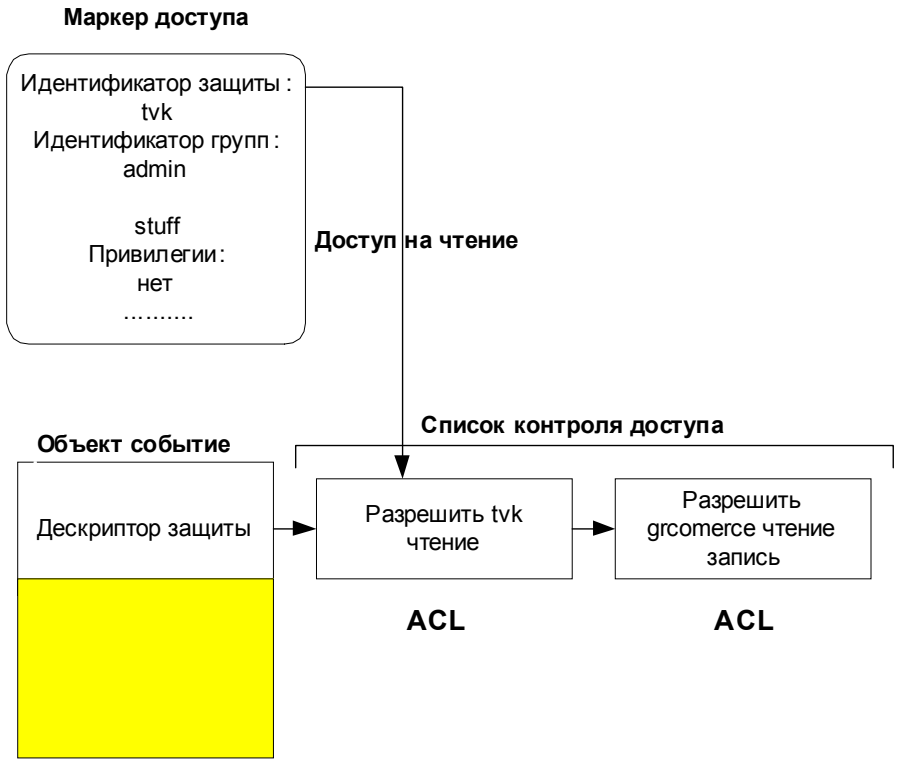
Кроме ACL, дескриптор защиты объектовсодержит поле, управляющее аудитом объекта. ***Аудит*** (***auditing***) - способность системы защиты шпионить за выбранными объектами и их пользователями и генерировать сообщения или оповещения, если кто-либо пытается выполнить операцию, использование которой с данным объектом запрещено. Например, аудит попыток чтения или изменения

системного файла. Если кто-то пытается модифицировать заданный файл, то система защиты помещает сообщение в журнал аудита с SID пользователя.

Как все это работает вместе

Маркер доступа идентифицирует процесс и его потоки для ОС, а дескриптор защиты содержит информацию о том, какие процессы имеют доступ к объекту. Когда поток открывает описатель объекта, диспетчер объектов и подсистема защиты сопоставляют эту информацию, чтобы определить, следует ли предоставить вызывающему потоку запрашиваемый им описатель (см. Рис. 8Проверка прав доступа).

Рисунок 3.8 Проверка прав доступа



Проверяя ACL, подсистема защиты просматривает список, начиная с первого ACE. Когда она находит идентификатор защиты вызывающего или его группы, она останавливает поиск и проверяет, разрешает ли данный ACE доступ запрашиваемого типа. Если ACL, разрешающий доступ, найден, то поиск прекращается и вызывающему возвращается описатель. Если подсистема достигает конца списка, не найдя идентификатор защиты вызывающего или его группы, то доступ запрещается.

Делать проверку при всяком использовании описателя процессом неэффективно, так как ACL может содержать много записей, а процесс может за время работы осуществлять доступ ко многим объектам. Кроме того, одновременно могут быть активными несколько процессов. Поэтому проверка осуществляется только при открытии описателя, а не при всяком его использовании. Когда объекты использует сама ОС используя указатели проверка прав доступа не производится.

После того, как процесс успешно открыл описатель, предоставленные ему права доступа не могут быть отозваны подсистемой защиты, даже если изменился ACL объекта. Последнее потребовало бы проверки прав доступа при всяком использовании описателя, а это могло бы значительно снизить производительность, особенно для объектов с длинным ACL.

ЛАБОРАТОРНАЯ РАБОТА 3

ОБЪЕКТЫ (ПРОГРАММИРОВАНИЕ)

Теория

Без четкого понимания, что такое объекты не стать профессионалом в области разработки Windows программ. Объекты используются ОС (исполнительной системой и ядром) и приложениями для управления различными ресурсами: процессами, потоками, файлами и т.д.

Объектами управляет ***диспетчером объектов*** (**Object Manager**) исполнительной системы. Исполнительная система создает объект, когда она выделяет ресурсы процессу. Приложение не может получить прямой доступ к объекту, а только через соответствующий интерфейс. Кроме того, объект размещается в системной области и пользовательские программы не могут напрямую обращаться к памяти, в которой объект расположен.

Что такое объект ?

Как разработчик Windows-приложений, вы будете постоянно выполнять различные операции с объектами. Типы используемых объектов частично уже упоминались, но приведем некоторые из них еще раз:

- **Object Directory** - тип объекта, который включает набор других объектов. Исполнительная система использует этот тип для организации набора объектов.
- **Symbolic link** - ссылка на другой объект по его символическому имени. Этот тип используется для поддержки такой возможности.
- **Process** - тип объекта, который представляет процесс.
- **Thread** - тип объекта, который представляет поток.
- **Section** - тип объекта, который используется для реализации разделяемой памяти, используемой в адресных пространствах процессов.
- **File Port** - тип объекта, который представляет описатель файла, когда файл открыт для использования.
- ·**Access Token** - тип объекта, который использует ***Security Reference, Monitor Local Security Authority*** (***LSA***) для аутентификации пользователя и другие части системы защиты.
- ·**Event** - класс объектов, которые могут перехватывать определенное ***событие*** (***event***) в системе, таким образом, что другие части системы смогут выполнить определенные действия, когда они уверены, что событие произошло. Объект событие это основа для многих операций синхронизации.
- **Semaphore** - классические **семафоры** (***Dijkstra***).
- **Mutex** - тип объекта для синхронизации, используемый для взаимно исключающего доступа к критическим секциям.
- **Timer** - тип объекта, который используется для уведомления потока о том, что заданное время истекло.

Каждый объект содержит данные, определяемые типом, и стандартный ***заголовок*** (***title***) с которым работает **диспетчер объектов** (***Object Manager***). В заголовке содержится имя объекта, атрибуты безопасности и т.д.

Каждый объект - это просто блок памяти, выделенный в системной области памяти. Этот блок представляет структуру данных, в элементах которой содержится информация об объекте. Некоторые элементы (дескриптор защиты, счетчик числа открытых описателей и д.р.) присутствует во всех объектах (заголовок), но большая часть специфична для объектов конкретного типа.

Приложение не может найти и модифицировать содержимое объектов. Для этого в Windows предусмотрен набор функций. Мы получаем доступ к объектам только через эти функции. Когда вы вызываете функцию, создающую объект, она возвращает описатель, идентифицирующий созданный объект. Далее этот описатель может использовать любой поток вашего процесса, чтобы сообщить системе какой объект вас интересует.

Для надежности ОС Microsoft сделала так, чтобы значения описателей зависели от конкретного процесса.

Таблица описателей объектов

При инициализации процесса система создает в нем таблицу описателей объектов. Сведения о структуре этой таблицы плохо документированы, но квалифицированный программист должен понимать, как устроена таблица описателей объектов в процессе. Как видно из таблицы (*Таблица 3.2*). Структура таблицы описателей объектов процесса, это просто массив структур данных. Каждая структура содержит указатель на какой-нибудь объект, маску доступа и некоторые флаги.

Таблица 3.2 Структура таблицы описателей объектов процесса

Индекс	Указатель на блок памяти объекта ядра	Маска доступа	Флаги
1	0x????????	0x????????	0x????????
2	0x????????	0x????????	0x????????
...	...	...	...

Создание объекта

Когда процесс создается, таблица описателей еще пуста. Но стоит одному из его потоков вызвать функцию, создающую объект, как диспетчер объектов выделяет для этого объекта блок памяти и инициализирует его. Дальше диспетчер объектов просматривает таблицу описателей процесса и ищет в ней первую свободную запись. Поскольку таблица еще пуста, то будет инициализирована структура с индексом 1. Указатель будет установлен на внутренний адрес структуры объекта, маска доступа - на доступ без ограничений.

Вот некоторые функции, создающие объекты ядра:

```
HANDLE CreateThread(
   PSECURITY_ATTRIBUTES psa,
   DWORD dwStackSize,
   PTHREAD_START_ROUTINE pfnStartAddr,
   PVOID pvParam,
   DWORD dwCreationFlags,
```

```
   PDWORD pdwThreadId);
HANDLE CreateFile(
   PCTSTR pszFileName,
   DWORD dwDesiredAccess,
   DWORD dwShareMade,
   PSECURITY_ATTRIBUTES psa,
   DWORD dwCreationDistribution,
   DWORD dwFlagsAndAttributes,
   HANDLE hTemplateFile);
HANDLE CreateFileMapping(
   HANDLE hFile,
   PSECURITY_ATTRIBUTES psa,
   DWORD flProtect,
DWORD dwMaximumSizeHigh,
DWORD dwMaximumSizeLow,
   PCTSTR pszFileName);
HANDLE CreateSemaphore(
   PSECURITY_ATTRIBUTES psa,
   LONG lInitialCount,
   LONG lMaximumCount,
   PCTSTR pszFileName);
```

Все функции, создающие объекты, возвращают описатели, которые привязаны к конкретному процессу и могут быть использованы в любом потоке данного процесса. Значение описателя представляет собой индекс в таблице описателей процесса, и таким образом идентифицирует место, где хранится объект. В Windows 2000 это значение определяет не индекс, а байтовое смещение нужной записи от начала таблицы.

Всякий раз, при вызове функции, которая в качестве аргумента использует описатель объекта, вы передаете ей значение, возвращаемое одной из **Create<Class>** функций. При этом функция смотрит в таблицу описателей объектов вашего процесса и считывает адрес нужного объекта.

Если вы передаете неверный описатель, то функция завершается с ошибкой и **GetLastError** возвращает **ERROR_INVALID_HANDLE**. Это может быть связано с тем, что описатели представляют собой индексы в таблице объектов процесса и недействительны в других процессах.

Если объект создать не удалось, то обычно возвращается **NULL (0)**. Это может произойти при нехватке памяти или проблем с защитой. Но некоторые функции возвращают в таком случае не **NULL**, а **INVALID_HANDLE_VALUE (-1)**. Например, если **CreateFile** не сможет открыть указанный файл, то она вернет **INVALID_HANDLE_VALUE**. Поэтому будьте очень внимательны при проверке значений, возвращаемых функциями, создающими объекты.

```
HANDLE hMutex = CreateMutex(…);
If ( hMutex == INVALID_HANDLE_VALUE )
{
// этот код не будет выполнен никогда
// при ошибке CreateMutex возвращает NULL
}
HANDLE hFile = CreateFile(…);
If (hFile == NULL)
{
// этот код не будет выполнен никогда
// при ошибке CreateFile возвращает INVALID_HANDLE_VALUE
}
```

Закрытие объекта

Независимо от того, как именно вы создали объект, по окончании работы с ним его нужно закрыть вызвав **CloseHandle**:

```
BOOL CloseHandle(HANDLE hObj);
```

Эта функция сначала проверяет таблицу описателей вызывающего процесса, чтобы убедиться, идентифицирует ли переданный ей описатель объект, к которому этот процесс действительно имеет доступ. Если передан правильный индекс, то система получает адрес структуры данных объекта и уменьшает в этой структуре счетчик открытых описателей; как только счетчик будет равен 0, объект будет удален из памяти.

Если описатель неверен, то происходит одно из двух:

- в нормальном режиме **CloseHandle** возвращает **FALSE**, а **GetLastError** - **ERROR_INVALID_HANDLE**;
- в режиме отладки система просто уведомляет об ошибке.

Перед самым возвратом управления **CloseHandle** удаляет соответствующую запись из таблицы описателей. После этого данный описатель недействителен в вашем процессе, и использовать его нельзя. После вызова **CloseHandle** вы больше не получите доступ к этому объекту ядра; но если счетчик числа открытых описателей не обнулен, то объект останется в памяти. Это означает, что объект используется другим процессом или процессами. Когда и они завершат с ним работу, вызвав **CloseHandle**, он будет уничтожен.

Учет объектов

Объекты принадлежат системе, а не процессу. Например, если ваш процесс создает объект, а затем завершается, объект может быть не уничтожен. Если созданный вами объект используется другим процессом, то система запретит его разрушение до тех пор, пока от него не откажется и тот процесс.

Системе известно, сколько процессов использует конкретный объект, так как в заголовке каждого объекта имеется счетчик открытых описателей. В момент создания объекта счетчику присваивается 1. Когда к существующему объекту обращается другой процесс, счетчик увеличивается на 1. А когда какой-то процесс завершается, все его счетчики автоматически уменьшаются на 1. Как только счетчик какого-либо объекта обнуляется, этот объект уничтожает.

Защита объектов

Вызовы функций **Create<Class>** или **Open<Class>** использовались для запроса системных ресурсов, для того, чтобы объект был размещен, а описатель возвращен вызывающему потоку. Механизм защиты диспетчера объектов требует, чтобы при вызове этих функций были заданы права доступ. Существует набор общих прав доступа для всех объектов (чтение и запись), также есть типозависимые права (например, отложить доступ к вновь созданному процессу). При вызове **Open<Class>** *диспетчер объектов* проверяет права доступа, используя **SECURITY_DESCRIPTOR** и если возможно, предоставляет желаемый доступ.

Механизм безопасности основан на возможности аутентифицировать пользователя, который запустил программу. Процесс WINLOGIN аутентифицирует пользователей, когда они регистрируются в системе. Когда *диспетчера объектов* создает объект, он создает дескриптор безопасности, который включает идентификатор владельца, дискретный и системный списки контроля доступа. Дискретный список контроля доступа это список процессов и их прав для объекта, а системный список контроля доступа это список процессов и их прав, которые должны быть записаны в журнал безопасности при использовании для последующего аудита.

Если доступ разрешен, то Security Reference Monitor возвращает точный набор прав доступа, которые гарантируются вызывающему процессу и *диспетчера объектов* сохраняет эти права как часть описателя объекта в таблице описателей объектов процесса. Когда поток процесса использует описатель, его права доступа к данному объекту сравниваются с гарантированным доступом, перед тем как к объекту будет предоставлен запрашиваемый доступ.

В лабораторной работе 2 мы уже использовали **CreateProcess** для создания нового процесса.

```
BOOL CreateProcess(
…
LPSECURITY_ATTRIBUTES lpProcessAttributes,
// атрибуты безопасности процесса ( указатель )
LPSECURITY_ATTRIBUTES lpThreadAttributes,
```

```
// атрибуты безопасности потока ( указатель )
…
);
```

lpProcessAttributes и **lpThreadAttributes** - это указатели на структуру данных **SECURITY_ATTRIBUTES**.

```
typedef struct_SECURITY_ATRIBUTES { //sa
DWORD nLength;
LPVOID lpSecurityDescriptor;
BOOL bInheritHandle;
} SECURITY_ATTRIBUTES;
```

lpSecurityDescriptor - это указатель на структуру **SECURITY_DESCRIPTOR**. Она не документирована, но ее поля можно прочитать или установить используя Win32 API (см. в **MSDN SECURITY_DESCRIPTOR**). Функции Win32 API **GetSecurityDescriptorControl**, **SetSecurityDescriptorDacl** и другие позволяют установить значения полей в **SECURITY_DESCRIPTOR**.. Таким образом, вызывающий поток может специфицировать желаемые права доступа, которые будут назначены объекту или использовать аутентифицированный доступ для существующего объекта.

Значение по умолчанию для атрибутов безопасности (в вызове **CreateProcess**) - **NULL**. Это означает, что структура **SECURITY_ATTRIBUTES** при вызове не передается. ***Диспетчер процессов*** (***Process Manage***r) и ***диспетчер объектов*** (***Object Manager***) интерпретируют **NULL** следующим образом: процесс ребенок (или поток) будет использовать **SECURITY_DESCRIPTOR** для существующего объекта или системное значение по умолчанию, если создается новый объект.

Если вы хотите, то можете ограничить доступ ребенка к его собственному объекту-процессу. Для этого необходимо создать **SECURITY_ATTRIBUTE** и выполнить вызов, чтобы список контроля доступа не давал возможности ребенку ссылаться на объект. Это необычно когда вы ограничиваете доступ ребенка к его собственному объекту процессу. Один и тот же механизм атрибутов безопасности используется для всех объектов исполнительной системы файлов, секций и т.д.

Итак, объекты можно защищать, используя ***дескриптор защиты*** (***security descriptor***), который описывает, кто создал объект и кто имеет права доступа к нему.

Почти все функции, создающие объекты, принимают указатель на структуру **SECURITY_ATTRIBUTES**. Рассмотрим еще один пример.

```
HANDLE CreateFileMapping(
HANDLE hFile,
PSECURITY_ATTRIBUTES psa,
DWORD flProtect,
DWORD dwMaximumSizeHigh,
DWORD dwMaximumSizeLow,
PCTSTR pszFileName);
```

Чтобы ограничить доступ к созданному объекту, создадим дескриптор защиты и инициализируем структуру **SECURITY_ATTRIBUTES** следующим образом:

```
SECURITY_ATTRIBUTES sa;
sa.nLength = sizeof(sa);
sa.lpSecurityDescriptor = pSD; // адрес инициализированной SD
sa.bInheritHandle = FALSE;
HANDLE hFileMapping = CreateFileMapping(INVALID_HANDLE_VALUE, &sa,
   PAGE_READWRITE, 0, 1024, "MyFileMapping");
```

Для получения доступа к существующему объекту, необходимо указать какие операции вы собираетесь проводить с ним. Например, если мы хотим считать данные из существующей проекции файла, то надо вызвать функцию **OpenFileMapping** так:

```
HANDLE hFileMapping=OpenFileMapping(FILE_MAP_READ, FALSE, "MyFileMapping");
```

Передавая первым параметром **FILE_MAP_READ**, мы просим предоставить доступ к проекции файла для чтения. Функция **OpenFileMapping** прежде чем вернуть описатель, проверит тип защиты объекта. Если доступ разрешен, то функция возвратит действительный описатель, а если в доступе отказано, то **NULL**, а вызов **GetLastError** вернет код ошибки **ERROR_ACCESS_DENIED (5)**.

Работа с объектами

Когда в лабораторной работе 2 ваш поток использовал **CreateProcess** и **CreateThread** для создания другого процесса или потока, он передавал имя ***диспетчеру объектов*** (***Object Manager***). При этом возвращались ***описатель*** (***HANDLE***) и идентификатор. В случае **CreateProcess** описатели нового процесса и нового потока, а также их идентификаторы заносятся в структуру данных **PROCESS_INFORMATION**. Функция **CreateThread** также возвращает описатель созданного потока и его идентификатор. После того как пользовательский поток получил описатель, он может передавать его исполнительной системе как параметр для других запросов. Например, после того как один поток создал другой поток, он передает описатель обратно исполнительной системе, закрывая описатель потока.

```
ChildThreadHandle = CreateThread(…);
…
CloseHandle(ChildThreadHandle);
```

Когда диспетчеру объектов обрабатывает первый вызов, который ссылается на объект, он:

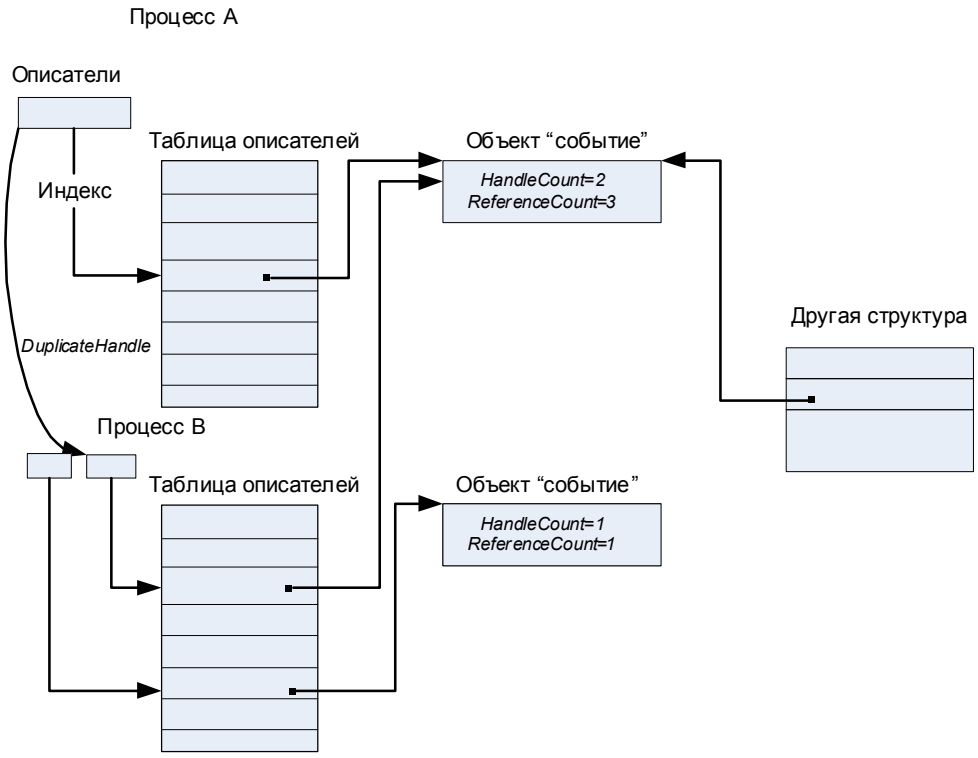
- добавляет имя объекта к набору используемых имен объектов;
- создает объект исполнительной системы с телом и заголовком;
- инициализирует поля заголовка;
- передает объект другим компонентам исполнительной системы, для заполнения тела типозависимой информацией.

Исполнительная система может определить, что такой объект уже существует, так как другой процесс (или другой поток) его уже создал. В этом случае счетчик открытых описателей в заголовке объекта увеличивается на 1, указывая, что теперь два описателя ссылаются на этот объект. Операции "открыть" приводят к увеличению значения счетчика на 1, а "закрыть" к уменьшению значения счетчика на 1. Когда значение счетчика становится равным 0, это означает, что больше не существует пользовательских описателей, которые ссылаются на этот объект. В этом случае диспетчеру объектов удаляет имя объекта из пространства имен.

Кроме того, компоненты исполнительной системы могут также ссылаться на объект, но для этого им нет необходимости использовать описатель (они могут использовать адрес объекта, так как они функционируют в режиме ядра). Поэтому заголовок объекта содержит также счетчик ссылок для хранения числа всех ссылок на объект (*Рисунок 3.9*).

Операции "открыть" в пользовательском режиме или в режиме ядра приводят к увеличению значения счетчика на 1, а "закрыть" к уменьшению значения счетчика на 1. Когда значение счетчика становится равным 0, это означает, что объект не используется никакими программными компонентами (пользовательскими или ядра) и можно освободить память, которую он занимал.

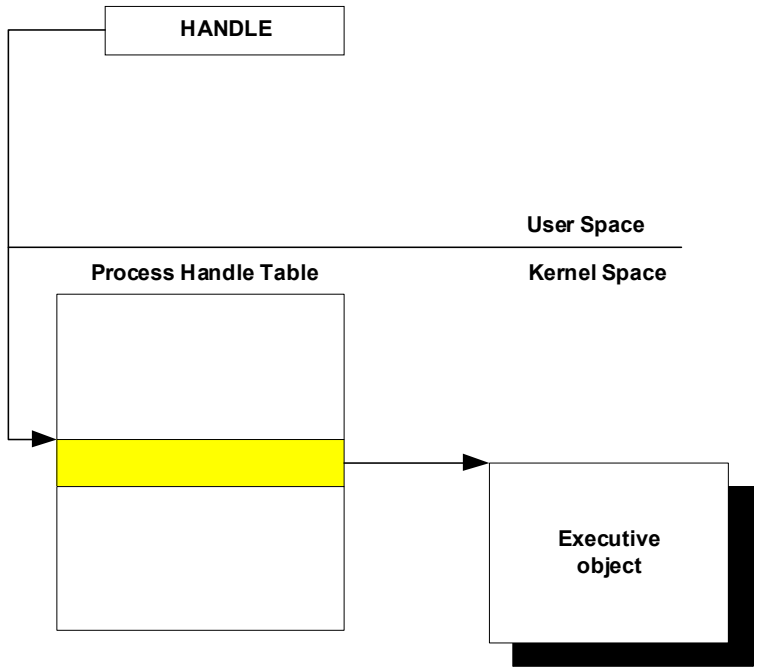
Рисунок 3.9 Счетчики описателей и ссылок



Какие отношения существуют между объектом ядра, описателем объекта и его идентификатором? Как показано на рисунке (*Рисунок 3.10*), описатели и таблица описателей создаются исполнительной системой и ядром и хранится в области памяти ядра, поэтому пользовательский поток не может получить к ним прямой доступ. Описатель - это 32-битная ссылка на объект, определенная для процесса. Это смещение в таблице описателей объектов процесса. Когда исполнительной системе нужно создать описатель объекта, она сначала ищет в таблице описателей процесса свободную строку. Затем заносит в первое поле адрес объекта в области ядра, во второе разрешения на доступ и т.д. Таким образом, возвращаемый описатель это индекс в таблице описателей объектов процесса, расположенной в области ядра. Описатель доступен только потокам, которые существуют в том же адресном пространстве.

Если необходимо идентифицировать объект в разных адресных пространствах, то вместо описателя можно использовать идентификатор.

Рисунок 3.10 Описатели и таблица описателей



Объект WaitableTimer

В этой лабораторной работе надо будет использовать объект ***ожидаемый таймер*** (***WaitableTimer***) для управления поведением потока. В лабораторной работе 2 предлагалось использовать **`GetSystemTime`** и разделяемую переменную **`runFlag`** для определения, когда остановить набор потоков. Один поток читал системное время и определял, не пора ли завершить работу, и если нет, то он блокировал себя на определенное время, а затем просыпался и снова проверял. Попробуем сделать лучше, используя ожидаемый таймер. Этот тип объекта умеет периодически сообщать вашему процессу, что заданный интервал времени истек. Точность 100 ns.

```
HANDLE CreateWaitableTimer(
LPSECURITY_ATTRIBUTES lpTimerAttributes,
// указатель на атрибуты безопасности
BOOL bManualReset, // флаг ручного перезапуска
LPCTSTR lpTimerName // указатель на имя объекта таймера
);
```

Атрибуты безопасности используются такие же, как в **`CreateProcess`** и **`CreateThread`**. Задавая **`lpTimerName`**, можно создать именованный объект или получить описатель уже существующего объекта (открыть). Если объект с таким именем уже существует, то **`GetLastError`** вернет **`ERROR_ALREADY_EXISTS`**. Параметр **`bManualReset`** контролирует число потоков, которые получат сигнал от ожидаемого таймера, когда он его

пошлет. Если он **TRUE**, то все потоки, которые ждут таймер, получат уведомление, в противном случае только один. Функция возвращает описатель таймера.

Поток может получить описатель существующего таймера, используя **OpenWaitableTimer**.

```
HANDLE OpenWaitableTimer(
DWORD dwDesiredAccesss, // флаг доступа
BOOL bInheritFlag, // флаг наследования
LPCTSTR lpTimerName // указатель на имя объекта таймера
);
```

Используя **dwDesiredAccess**, процесс может задать то, как он собирается использовать описатель таймера. Вы можете только получать уведомление от таймера, устанавливать период между посылаемыми уведомлениями или иметь полный доступ ко всем функциям таймера.

После того, как таймер создан, он должен быть установлен, перед тем как начнет выдавать уведомления. Для этого используется функция **SetWaitableTimer**.

```
BOOL SetWaitableTimer(
HANDLE hTimer, // описатель таймера
Const LARGE_INTEGER *pDueTime,
// когда таймер перейдет в состояние signaled
LONG lPeriod, // периодический временной интервал
PTIMERAPCROUTINE pfnCompletionRoutine,
// указатель на процедуру (APC)
LPVOID lpArgToCompletionRoutine,
// данные, передаваемые в процедуру (APC)
BOOL fResume // флаг состояния
);
```

htimer - это описатель, который возвращают функции **CreateWaitableTimer** и **OpenWaitableTimer** (при его создании или открытии).

pdueTime - это число в 64-битовом формате **FILETIME**.

```
typedef struct_FILETIME { //ft
DWORD dwLowDateTime;
// это младшие 32 бита времени в системном формате
DWORD dwHighDateTime;
// это старшие 32 бита
} FILETIME;
```

Вы можете задать **pDueTime** как относительное или абсолютное время, отличить относительное время от абсолютного можно установив 64-битовое время отрицательным. **lperiod** задает время в милисекундах между уведомлениями. Если задано 0 значение, то таймер пошлет уведомление только один раз.

Следующие два параметра **pfnCompletionRoutine** и **lpArgToCompletionRoutine** здесь не рассматриваются. Флаг **fResume** необходим для особого режима использования компьютера, в случае если компьютер завис, то если значение флага **TRUE**, при получении уведомления он будет перезагружен.

Теперь рассмотрим использование функции **WaitForSingleObject**. Когда поток вызывает эту функцию, он будет заблокирован, пока не придет уведомление от указанного объекта. Поток может создать таймер, установить его и затем ждать от него уведомления, вызвав **WaitForSingleObjec**t.

```
DWORD WaitForSignleObject(
HANDLE hHandle; // описатель ожидаемого объекта
DWORD dwMilliseconds; // тайм-аут в милисекудах
);
```

У нас **hHandle** - это описатель таймера. **dwMilliseconds** определяет максимальный промежуток времени, который поток желает ждать пока истечет время. Вы можете использовать **GetLastError**, чтобы посмотреть, что возвращает функция:

- **WAIT_OBJECT_O** - если послано уведомление;
- или **WAIT_TIMEOUT** - истек тайм-аут.

Вы можете также задать **INFINITE** для ожидания уведомления без тайм-аута.

Скелет кода с использованием таймера

```
#define _WIN32_WINNT 0x0400

#include <windows.h>
#include <stdio.h>

#define _SECOND 10000000

void main( void )
{
HANDLE wTimer;
__int64 endTime;
LARGE_INTEGER quitTime;
SYSTEMTIME now;
```

```
wTimer = CreateWaitableTimer(NULL,FALSE,NULL);
endTime = -5 * _SECOND;
quitTime.LowPart = (DWORD) (endTime & 0xFFFFFFFF);
quitTime.HighPart = (LONG) (endTime >> 32);
SetWaitableTimer(wTimer,&quitTime,0,NULL,NULL,FALSE);
GetSystemTime(&now);
printf("System Time %d %d %d\n",now.wHour,now.wMinute,now.wSecond);
WaitForSingleObject(wTimer,INFINITE);
printf("Waitable Timer sent signal\n");
GetSystemTime(&now);
printf("System Time %d %d %d\n",now.wHour,now.wMinute,now.wSecond);
CloseHandle( wTimer );
}
```

Чему нужно научиться ?

Необходимо научиться работать с объектами.

Задания

Уровень 1 (A)

Написать программу, которая использует ожидаемый таймер (**`WaitableTimer`**), чтобы остановить себя через K секунд после старта, где K - параметр командной строки.

Уровень 2 (A)

Модифицировать программу первого уровня, чтобы она создавала N фоновых процессов, каждый запускал бы программу, которая заканчивалась бы в случайное время по своему собственному усмотрению (N - параметр командной строки). А после того, как истекут K секунд (K - параметр командной строки), она уничтожала процессы, которые не закончились сами. Если все фоновые процессы закончились, то главная программа сама выполняется K секунд, а потом заканчивается.

Уровень 3 (A)

Поработайте с правами доступа. Убедитесь в том, что проверка прав доступа осуществляется только при открытии описателя, а не при всяком его использовании. После того, как процесс успешно открыл описатель, предоставленные ему права доступа не отзываются подсистемой защиты, даже если изменился ACL объекта.

Используйте функции: **`GetEffectiveRightsFromAcl`**, **`AddAccessAllowedAce`**, **`SetSecurityInfo`**, **`SetNamedSecurityInfo`** и т.д.

Скелет кода фоновой программы

Хотим создать процесс, который может закончиться или не закончиться по своему собственному усмотрению.

```
#include <windows.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
printf("Client %s beginning to run\n",argv[1]);
while(TRUE)
{
if(getc(stdin)=='y') break;
getc(stdin);
}
return 0;
}
```

Для выполнения задания вам может понадобиться **TerminateProcess**:

```
BOOL TerminateProcess(
HANDLE hProcess, // описатель процесса
UINT uExitCode // код выхода для процесса
);
```

Один процесс может закончить другой процесс, но он должен иметь на это соответствующий допуск (**PROCESS_TERMINATE**) для этого описателя.

Глава

4

Синхронизация

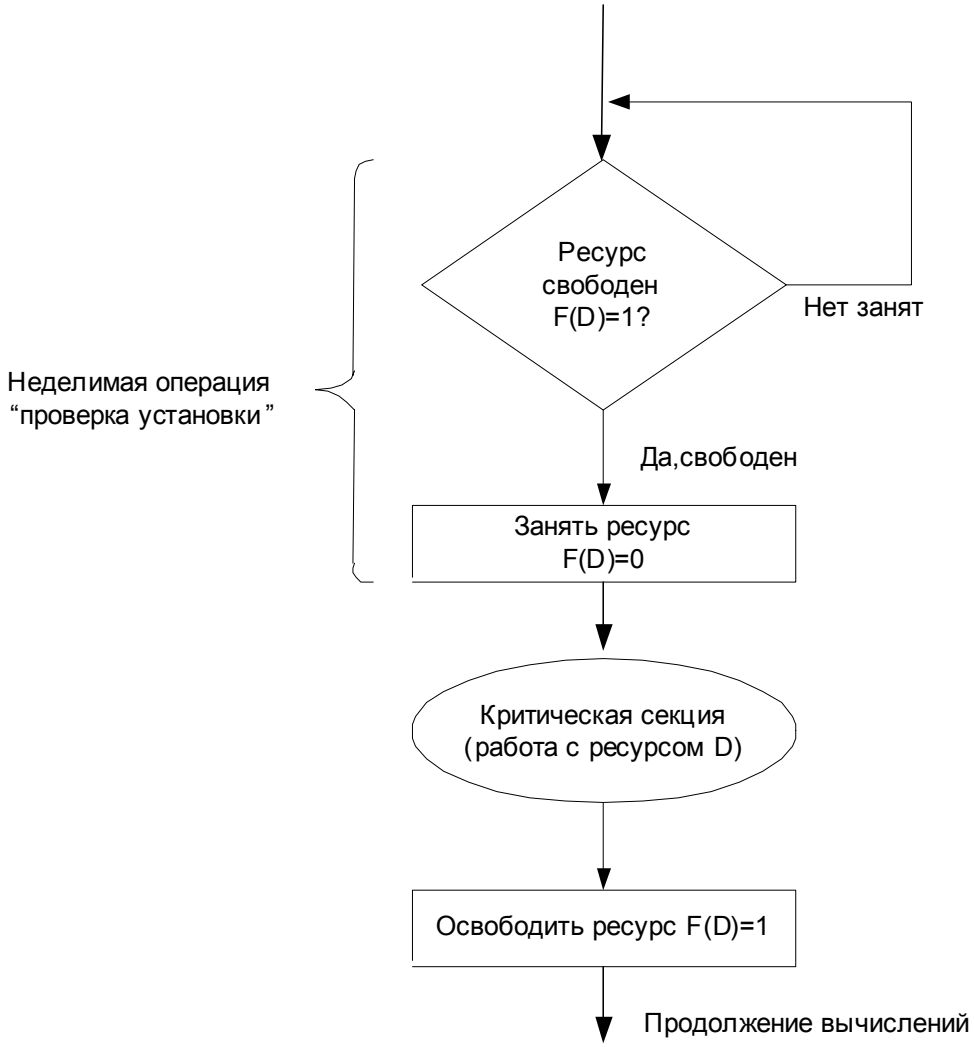
Основные понятия и определения

Смысл ***взаимоисключения*** (***mutual exclusion***) заключается в следующем: в каждый момент времени к ресурсу может обращаться только один поток. Взаимоисключение нужно использовать тогда, когда ресурс не предназначен для разделения или когда разделение может иметь непредсказуемые последствия. Например, если два потока будут посылать данные в порт принтера, то получится не документ, а адская смесь.

Участки кода, в которых происходит обращение к неразделяемым ресурсам, называются ***критическими секциями*** (***critical sections***) В критической секции единовременно может находиться только один поток. Пока один поток записывает данные в файл, обновляет базу данных или модифицирует общую переменную, доступ к этому ресурсу для других потоков запрещен.

Каждому набору критических данных ставится в соответствие двоичная переменная, которой поток присваивает значение 0, когда он входит в критическую секцию, и значение 1, когда он ее покидает. На рисунке (*Рисунок 4.1*) показан фрагмент алгоритма потока, использующего для реализации взаимного исключения доступа к критическим данным D блокирующую переменную F(D). Перед входом в критическую секцию поток проверяет, не работает ли уже какой-нибудь поток с данными D. Если переменная F(D) установлена в 0, то данные заняты, и проверка циклически повторяется. Если же данные свободны (F(D) = 1), то значение переменной F(D) устанавливается в 0 и поток входит в критическую секцию. После того как поток выполнит все действия с данными, значение переменной F(D) снова устанавливается равным 1.

Рисунок 4.1 Использование блокирующих переменных



Нельзя прерывать поток между выполнением операций проверки и установки блокирующей переменной, это ***неделимая*** (***атомарная операция***). Если она началась, то не может быть прервана, пока не закончится.

Реализация взаимного исключения таким способом имеет существенный недостаток: в течение времени, когда один поток находится в критической секции, другой поток, которому требуется тот же ресурс, получив доступ к процессору, будет непрерывно опрашивать блокирующую переменную, бесполезно тратя выделяемое ему процессорное время, которое могло бы быть использовано для выполнения какого-нибудь другого потока. Для устранения этого недостатка во многих ОС предусматриваются специальные системные вызовы для работы с критическими секциями.

Обобщением блокирующих переменных являются так называемые ***семафоры Дийкстры***. Вместо двоичных переменных Дийкстра (Dijkstra) предложил использовать переменные, которые могут принимать целые неотрицательные значения. Такие переменные, используемые для синхронизации вычислительных процессов, получили название семафоров.

Для работы с семафорами вводятся два примитива, традиционно обозначаемых P и V. Пусть переменная S представляет собой семафор. Тогда действия V(S) и P(S) определяются следующим образом.

- **V(S)**: переменная S увеличивается на 1 единым действием. Выборка, наращивание и запоминание не могут быть прерваны. К переменной S нет доступа другим потокам во время выполнения этой операции.
- **P(S)**: уменьшение S на 1, если это возможно. Если S=0 и невозможно уменьшить S, оставаясь в области целых неотрицательных значений, то в этом случае поток, вызывающий операцию P, ждет, пока это уменьшение станет возможным. Успешная проверка и уменьшение также являются неделимой операцией.

Никакие прерывания во время выполнения примитивов V и P недопустимы.

В частном случае, когда семафор S может принимать только значения 0 и 1, он превращается в блокирующую переменную, которую по этой причине часто называют двоичным семафором.

Синхронизация в ОС

Взаимоисключение, важно для всех ОС, но особенно для ОС с *симметричной мультипроцессорной обработкой,* как Windows, в которой системный код, выполняемый на нескольких процессорах одновременно, разделяет глобальные структуры данных. В Windows поддержка механизмов, с помощью которых системный код может предотвратить одновременное изменение двумя потоками одной и той же структуры, возлагается на ядро. Оно предоставляет специальные примитивы взаимоисключения, используемые им и остальными компонентами исполнительной системы для синхронизации доступа к глобальным структурам данных.

Синхронизация ядра

Ядро должно гарантировать, что в каждый момент времени только один процессор выполняет код в критической секции. Критическими секциями ядра являются разделы кода, модифицирующие глобальные структуры данных, например очередь планировщика.

В момент обновления ядром глобальной структуры данных может произойти прерывание, процедура обработки которого изменяет ту же структуру. В однопроцессорных системах просто отключают прерывания на время доступа к глобальным данным. В Windows перед использованием глобального ресурса ядро временно маскирует прерывания, обработчики которых используют тот же ресурс. Это хорошо работает в однопроцессорных системах, но не годится для многопроцессорных систем.

Механизм, используемый ядром для взаимоисключения в многопроцессорных системах, называется ***спин-блокировкой*** (***spinlock***). *Спин-блокировка* - это блокирующий примитив, сопоставленный с какой-либо глобальной структурой данных. Перед входом в любую из критических секций ядро должно установить спин-блокировку на защищаемую структуру данных. Если спин-блокировка занята, ядро продолжает попытки установить ее, до тех пор, пока не получится. Ядро крутится (spin) в цикле, повторяя попытки до тех пор, пока не установит блокировку. Во многих архитектурах спин-блокировка реализуется аппаратно поддерживаемой командой test-and-set, которая проверяет значение переменной блокировки и устанавливает блокировку, выполняя одну атомарную команду.

Синхронизация в исполнительной системе

Компоненты исполнительной системы также нуждаются в синхронизации доступа к глобальным структурам данных в многопроцессорной среде. Вызывая функции ядра, исполнительная система может создать спин-блокировку, установить и снять ее.

Поскольку спин-блокировка означает фактическую остановку процессора, она применяется только в двух случаях:

- требуется непродолжительное обращение к защищенным ресурсам без взаимодействия с другим кодом;
- код критической секции нельзя выгрузить в страничный файл, он не ссылается на данные в подкачиваемой памяти, не вызывает внешние процедуры (включая системные сервисы) и не генерирует прерывания или исключения.

Эти противоречащие друг другу ограничения нельзя соблюсти одновременно и кроме того, исполнительная система должна поддерживать и другие механизмы синхронизации, а также предоставлять механизмы синхронизации пользовательскому режиму.

Ядро предоставляет исполнительной системе дополнительные механизмы синхронизации в форме синхронизирующих объектов ядра. Синхронизирующие объекты, видимые из пользовательского режима, включают объекты ядра. Синхронизационные механизмы исполнительной системы доступны программистам через Win32-функции **`WaitForSingleObject`** и **`WaitForMultipleObjects`**. Поток можно синхронизировать по следующим объектам: процесс, поток, событие, семафор, мьютекс, ожидаемый таймер или файл.

Ожидание

В любой момент, синхронизирующий объект находится в одном из двух состояний:

- свободном (*signaled*);
- или занятом (*non signaled*).

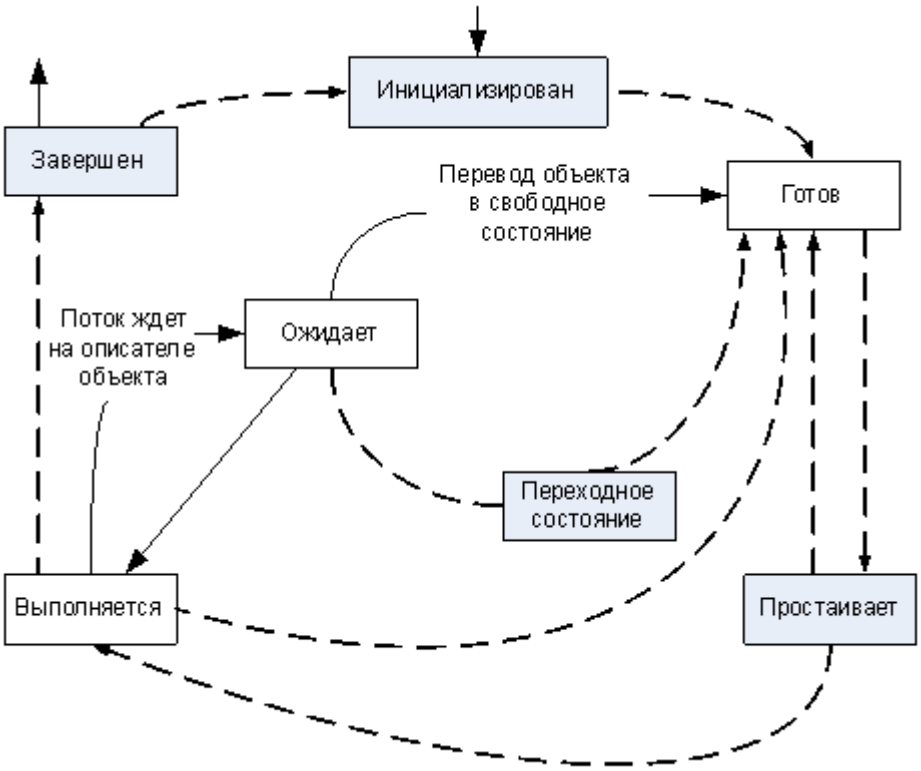
Поток синхронизируется с объектом, ожидая его освобождения. При этом ядро приостанавливает поток, удаляет его из очереди готовых к выполнению потоков и перестает учитывать его при планировании. Поток может ожидать на одном или нескольких объектах, а также указать, что ожидание следует прекратить, если объект или объекты не освободились в течение определенного времени.

Взаимосвязь синхронизации с диспетчеризацией потоков приведена на рисунке (*Рисунок 4.2*).

1. Поток пользовательского режима ждет на описателе объекта "событие" (ждет перехода этого объекта в свободное состояние).
2. Ядро изменяет состояние потока с "готов" на "ожидает" и добавляет его в список потоков, ждущих объект "событие".
3. Другой поток устанавливает (освобождает) объект "событие".
4. Ядро просматривает список потоков, ожидающих этот объект. Если условия ожидания какого-либо потока выполнены, ядро переводит его из состояния "ожидает" в состояние "готов". Если это поток с динамическим приоритетом, то ядро может повысить его приоритет.

5. Поток теперь готов к выполнению и если ядро обнаружит, что приоритет выполняемого в данный момент потока ниже, чем приоритет "дождавшегося" потока, то оно вытеснит поток более низким приоритетом и переключит контекст на наш поток.
6. Если вытеснение в данный момент невозможно, то дождавшийся поток будет стоять в очереди готовых к выполнению потоков.

Рисунок 4.2 Ожидание



Условия перехода в свободное состояние

Условия перехода в свободное состояние различны для разных объектов. Например, находится в занятом состоянии в течение всей своей жизни и переводится ядром в свободное состояние при завершении. Аналогично, ядро переводит объект "процесс" в свободное состояние в момент завершения последнего потока процесса. Таймер переводится в свободное состояние по истечении заданного времени.

Выбирая механизм синхронизации, нужно учитывать поведение синхронизирующих объектов. Когда объект переводится в свободное состояние, ожидающие его потоки обычно немедленно выходят из состояния ожидания (*Таблица 4.1*).

Таблица 4.1 Условия освобождения синхронизирующих объектов

Тип объекта	*Условие перехода в свободное состояние*	*Действие на ожидающие потоки*
Процесс	При завершении последнего потока	Все потоки дождались
Поток	При завершении данного потока	Все потоки дождались
Событие (уведомляющего типа)	При установке события потоком	Все потоки дождались
Событие (синхронизирующего типа)	При установке события потоком	Дождался один поток, и он занимает объект событие
Мьютекс	При освобождении объекта мьютекс потоком	Дождался один поток
Таймер (уведомляющего типа)	При истечении заданного времени или наступлении очередного момента срабатывания	Все потоки дождались
Таймер (синхронизирующего типа)	При истечении заданного времени или наступлении очередного момента срабатывания	Дождался один поток

Объекты, которые можно использовать для синхронизации, представлены на рисунке (*Рисунок 4.3*).

Синхронизация в пользовательском режиме

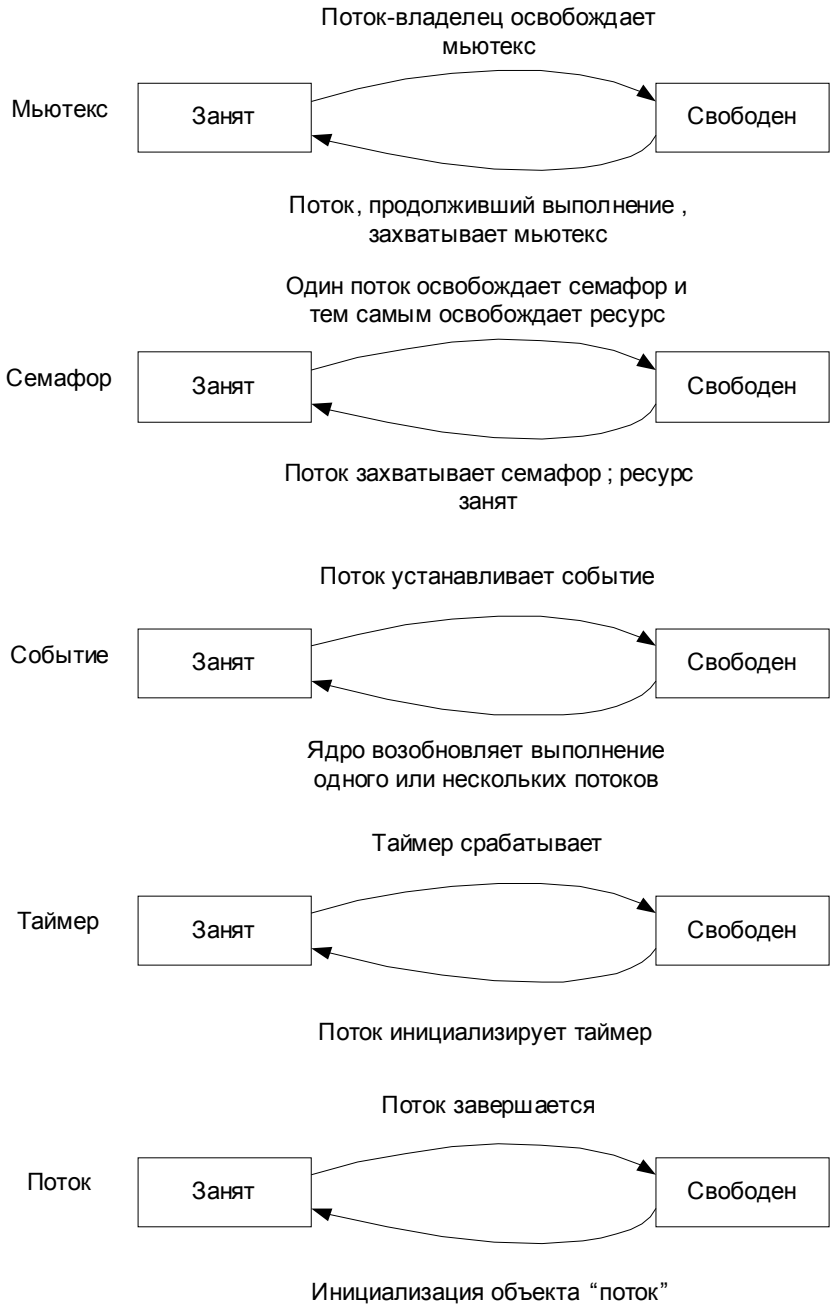
Windows лучше всего работает, когда все потоки занимаются каждый своим делом и не взаимодействуют друг с другом. Обычно так не бывает. Обычно поток выполняет определенную работу, о завершении которой хочет узнать другой поток.

Все потоки в системе могут иметь доступ к системным ресурсам. При этом нельзя предоставлять бесконтрольный доступ, иначе может получиться так, что один поток пишет в блок памяти, а другой этого же блока памяти что-то считывает. Получается, что вы читаете книгу, а в это время кто-то переписывает текст на открытой вами странице. Ничего хорошего из этого не выйдет.

Потоки взаимодействуют друг с другом в двух случаях:

- совместно используя разделяемый ресурс (чтобы его не разрушить);
- для уведомления других потоков о завершении каких-либо операций

Рисунок 4.3 Объекты, которые можно использовать для синхронизации



В Windows есть масса средств, упрощающих синхронизацию потоков. Только при этом необходимо учитывать, что точно спрогнозировать, в какой момент поток будет делать определенную работу крайне сложно. Мы не умеем думать асинхронно; только последовательно - обдумывая мысль за мыслью.

Единственный способ освоить синхронизацию потоков - действительно заняться этим на практике.

Атомарный доступ (семейство Interlocked-функций)

Синхронизация потоков связана с ***атомарным доступом*** (***atomic access***) - монопольный захват ресурса обращающимся к нему потоком. Рассмотрим пример:

```
// определим глобальную переменную g_x
long g_x = 0;

DWORD WINAPI ThreadFunc1(PVOID pvParam)
{
  g_x++;
  return 0;
}

DWORD WINAPI ThreadFunc2(PVOID pvParam)
{
  g_x++;
  return 0;
}
```

Мы объявили глобальную переменную **g_x** и инициализировали ее 0. Далее создадим два потока: **ThreadFunc1** и **ThreadFunc2**. Код этих функций идентичен: обе увеличивают значение глобальной переменной **g_x** на 1. Каким будет значение **g_x**, когда оба потока завершат работу? Допустим, что компилятор сгенерировал для строки **g_x++** следующий код:

```
MOV EAX, [g_x]    // значение из g_x помещается в регистр
INC EAX         // значение регистра увеличивается на 1
MOV [g_x], EAX   // значение из регистра помещается обратно в g_x
```

Если потоки будут выполнять этот код по очереди (сначала один, потом - другой), мы получим:

```
MOV EAX, [g_x]    // поток 1: в регистр помещается 0
INC EAX         // поток 1: значение регистра увеличивается на 1
MOV [g_x], EAX   // поток 1: значение 1 помещается обратно в g_x

MOV EAX, [g_x]    // поток 2: в регистр помещается 1
INC EAX         // поток 2: значение регистра увеличивается на 1
MOV [g_x], EAX   // поток 2: значение 2 помещается обратно в g_x
```

После выполнения обоих потоков значение **g_x** будет равно 2. Здорово! Но Windows - это среда, которая поддерживает многопоточность и вытесняющую многозадачность. Значит, процессорное время может быть в неопределенное время отнято у одного потока и передано другому. Тогда приведенный код может выполняться следующим образом:

```
MOV EAX, [g_x]    // поток 1: в регистр помещается 0
INC EAX         // поток 1: значение регистра увеличивается на 1

MOV EAX, [g_x]    // поток 2: в регистр помещается 1
INC EAX         // поток 2: значение регистра увеличивается на 1
MOV [g_x], EAX   // поток 2: значение 2 помещается обратно в g_x

MOV [g_x], EAX   // поток 1: значение 1 помещается обратно в g_x
```

Если код будет выполняться так, то значений **g_x** окажется равным 1, а не 2. Фактически, даже при сотне потоков, которые выполняют эту функцию, мы вполне можем получить в **g_x** все ту же 1. Кстати результаты будут зависеть от того, какой именно компилятор генерирует машинный код, как процессор выполняет этот код и сколько в машине процессоров. Это объективная реальность, в которой мы не в состоянии что-либо изменить.

Для решения этой проблемы нам нужен способ, гарантирующий приращение значения переменной на уровне атомарного доступа, т.е. без прерывания другими потоками. Для этого используется семейство **Interlocked**-функций. Все функции этого семейства манипулируют переменными на уровне атомарного доступа.

```
LONG InterlockedExchangeAdd(
  PLONG plAddend,
  LONG lIncrement);
```

Вы вызываете эту функцию, передавая адрес переменной типа **LONG**, и указываете добавляемое значение. **InterlockedExchangeAdd** гарантирует, что операция будет выполнена атомарно. Перепишем наш код:

```
// определим глобальную переменную g_x
long g_x = 0;

DWORD WINAPI ThreadFunc1(PVOID pvParam)
{
  InterlockedExchangeAdd(&g_x, 1);
  return 0;
}

DWORD WINAPI ThreadFunc2(PVOID pvParam)
{
  InterlockedExchangeAdd(&g_x, 1);
  return 0;
}
```

Теперь вы можете быть уверены, что конечное значение **g_x** будет равно 2.

Внимание!!! В любом потоке, где нужно модифицировать значение разделяемой переменной, следует пользоваться **`Interlocked`**-функциями и никогда не использовать стандартные операторы языка C.

```
LONG g_x;
…
g_x++;  // неправильный способ
…
InterlockedExchangeAdd(&g_x, 1); // правильный способ
```

Рассмотрим, как работают **`Interlocked`**-функции? Это зависит от того, какой процессор вы используете. Действует примерно такой алгоритм:

1. Устанавливается специальный битовый флаг процессора, указывающий, что даный адрес памяти сейчас занят.
2. Считывается значение из памяти в регистр.
3. Изменяется значение в регистре.
4. Если битовый флаг сброшен, операции повторяются с п.2, иначе - значение из регистра помещается обратно в память.

Вникать в детали реализации этих функций совершенно не обязательно. Главное, что они гарантируют монопольное изменение значений переменных независимо от того, как компилятор генерирует код и сколько процессоров у компьютера.

Еще один очень важный аспект, использования **`Interlocked`**-функций, состоит в том, что они выполняются очень быстро. Вызов такой функции обычно требует не более 50 тактов процессора, и при этом не происходит перехода из пользовательского режима в режим ядра (такой переход отнимает не менее 1000 тактов).

Как не надо делать

`Interlocked`-функции можно использовать, если необходимо монопольно изменить всего одну переменную. Но программы имеют дело со структурами данных. Чтобы получить доступ к таким структурам на атомарном уровне, необходимо использовать другие механизмы.

Рассмотрим, как это можно реализовать синхронизацию самостоятельно (*Рисунок 4.1*). Пусть поток синхронизирует себя с завершением какой-либо задачи в другом потоке, постоянно просматривая значение переменной, доступной обоим потокам.

```
BOOL g_fFinishedCalculation = FALSE;

volatile int WINAPI WinMain (…)
{
   CreateThread(…, RecalcFunc,…);
…
while(!g_fFinishedCalculation)
….
}

DWORD WINAPI RecalcFunc(PVOID pvParam)
{
   …
   g_fFinishedCalculation = TRUE;
   return 0;
}
```

Первичный поток (он исполняет **WinMain**) при синхронизации по событию - завершение функции **RecalcFunc**, никогда не впадет в спячку. Система будет выделять ему процессорное время за счет других потоков, которые могут заниматься чем-то гораздо более важным.

Переменная **g_fFinishedCalculation** может не стать **TRUE**, если у первичного потока более высокий приоритет, чем у потока, который выполняет функцию **RecalcFunc**. В этом случае система никогда не предоставит потоку **RecalcFunc** процессорное время, и он не выполнит оператор **g_fFinishedCalculation = TRUE**. Если бы мы не производили постоянный опрос (**while(!g_fFinishedCalculation)**), а просто отправили бы этот поток в спячку, то это позволило бы системеотдать его долю процессорного времени потоку с более низким приоритетом **RecalcFunc**.

Поэтому вместо опроса лучше пользоваться функциями, который переводят поток в состояние ожидания до освобождения нужного ему ресурса.

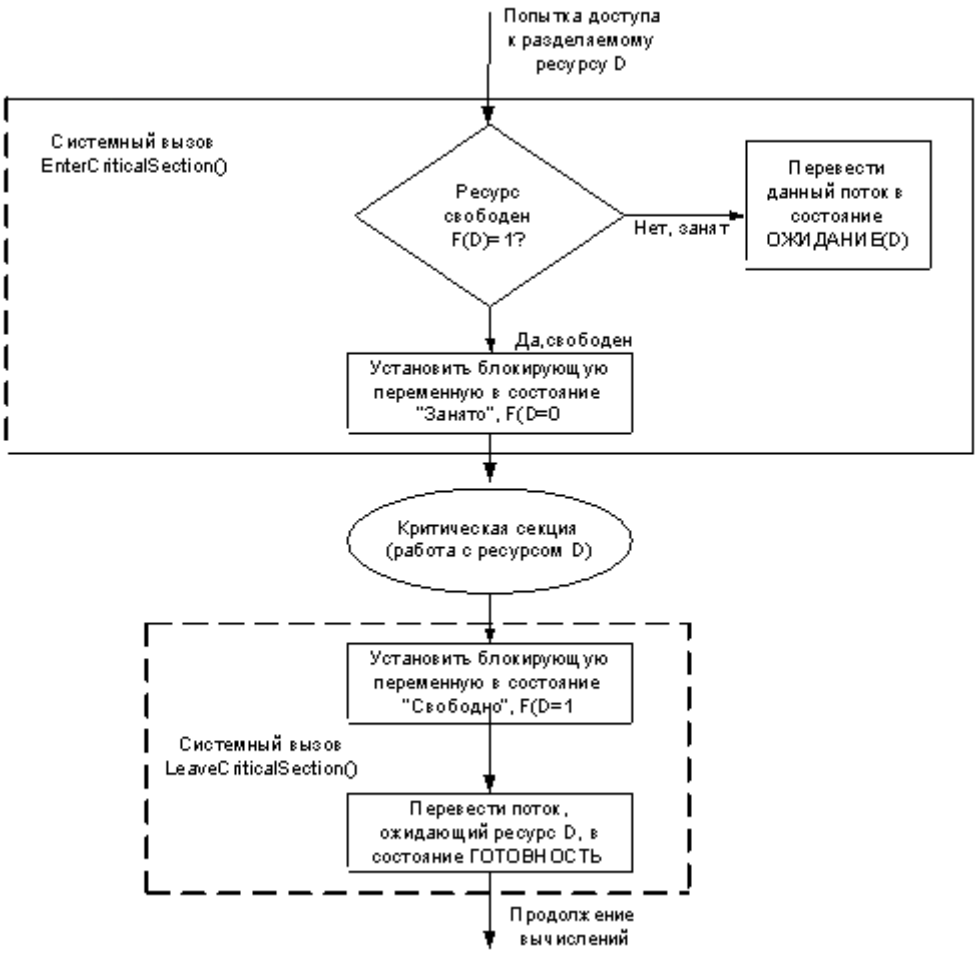
Когда поток хочет обратиться к разделяемому ресурсу или получить уведомление о некотором событии, он должен вызвать определенную функцию ОС и передать ей параметры, сообщающие, чего именно он ждет. Как только ОС обнаружит, что ресурс освободился или заданное событие произошло, эта функция вернет управление потоку, и он снова будет включен в число планируемых.

Пока ресурс не освободился или пока не произошло событие, система переводит поток в ждущий режим, исключая его из числа планируемых. ОС берет на себя роль агента, действующего в интересах спящего потока. Она выведет его из ждущего состояния, когда освободится нужный ресурс или произойдет событие.

Критические секции

Критическая секция (***critical section***) - это участок кода, требующий монопольного доступа к каким-то общим данным. Ее использование позволяет сделать так, чтобы единовременно только один поток получал доступ к этим данным (*Рисунок 4.4*). Система может в любой момент вытеснить ваш поток и назначить на процессор другой, но при этом ни один из потоков, которым нужны эти данные, не получит процессорное время до тех пор, пока ваш поток не выйдет за границы критической секции.

Рисунок 4.4 Реализация критической секции



Синхронизация с использованием объектов

Хотя механизмы синхронизации работают очень быстро, у них имеется ряд ограничений и во многих приложениях они не будут работать. Например, Interlocked-функции работают только с отдельными переменными и никогда не переводят поток в состояние ожидания. Эту задачу можно решить с помощью критических секций, но они подходят только для синхронизации потоков в рамках одного процесса. Кроме того, при использовании критических секций можно попасть в ситуацию взаимной блокировки потоков.

Рассмотрим, как синхронизовать потоки с помощью объектов. Такие объекты предоставляют гораздо больше возможностей, чем механизмы синхронизации в пользовательском режиме. Единственный их недостаток - меньшее быстродействие. Дело в том, что при вызове любой из таких функций, поток должен перейти из пользовательского режима в режим ядра, а это примерно 1000 процессорных тактов. А также необходимо учесть время на выполнение самой вызванной функции в режиме ядра.

Wait-функции

Функции ожидания используют описатель для доступа к объекту и проверяют его состояние:

· если объект находится в состоянии ***свободен*** (***singnaled***), то происходит возврат;

· если объект находится в состоянии ***занят*** (***nonsingnaled***), то функция ожидания блокирует вызвавший поток до тех пор, пока не возникнут определенные условия (объект перейдет в состояние свободен или истечет таймаут).

Возврат после вызова функции, опять делает поток ***выполняющимся*** (***runnable***) - он готов снова бороться за процессор. Таким образом, механизм синхронизации взаимодействует с механизмом планирования.

Что вызывает переход объекта из одного состояния в другое? В некоторых случаях это побочный эффект активности объекта, иногда результат определенных действий.

Wait-функции позволяют потоку в любой момент приостановиться и ждать освобождения какого-либо объекта. Чаще всего используется **`WaitForSingleObject`**.

События

События - это самые примитивный тип объектов. Они содержат стандартный заголовок (счетчик числа открытых описателей, … как и все объекты) и две булевы переменные. Одна это тип данного события, другая его состояние (свободен он или занят).

События просто уведомляют об окончании какой-либо операции. Объекты события бывают двух типов: со ***сбросом вручную*** (***manual-reset events***) и с ***авто-сбросом*** (***auto-reset events***). События первого типа позволяют возобновлять выполнение сразу нескольких ждущих потоков, а второго типа - только одного.

Объекты события обычно используются в том случае, когда какой-то поток выполняет инициализацию, а потом сигнализирует другому потоку, что тот может продолжить выполнение. Инициализирующий поток переводит объект событие в занятое состояние и приступает к работе. Закончив, он сбрасывает событие в свободное состояние. Тогда другой поток, который ждал перехода события в свободное состояние, пробуждается и вновь становится планируемым.

Объект ядра события создается функцией **`CreateEvent`**.

Ожидаемые таймеры

Ожидаемые таймеры (***waitable timers***) - это объекты, которые самостоятельно переходят в свободное состояние в определенное время или через регулярные промежутки времени. Ожидаемый таймер создается функцией **`CreateWaitableTimer`**.

Семафоры

Объекты ядра семафор используются для учета ресурсов. Семафор поддерживает два 32 битных значения со знаком: одно определяет максимальное число ресурсов (контролируемое семафором), а другое используется как счетчик текущего числа ресурсов.

Рассмотрим пример возможного использования семафоров. Допустим, мы разрабатываем серверный процесс, в адресном пространстве которого выделяется буфер для хранения клиентских запросов. Пусть размер этого буфера рассчитан максимум на 5 клиентских запросов. Если новый клиент пытается связаться с сервером, когда эти пять запросов еще не обработаны, то генерируется ошибка. Сервер занят и надо повторить попытку позже. При инициализации серверный процесс создает пул из 5 потоков, каждый из которых готов обрабатывать клиентские запросы по мере их поступления.

Когда запросов от клиентов еще нет, сервер не разрешает выделять процессорное время потокам в пуле. Но как только серверу поступает три клиентских запроса одновременно, три потока в пуле становятся планируемыми, и система начинает выделять им процессорное время. Используем для слежения за ресурсами и планированием потоков семафор. Зададим максимальное число ресурсов равным 5 (соответствует размеру буфера). Счетчик текущего числа ресурсов вначале равен 0, так как клиенты еще не выдали ни одного запроса. Значение счетчика увеличивается на 1 в момент приема очередного клиентского запроса и уменьшается на 1, когда запрос передается на обработку одному из серверных потоков в пуле.

Правила, определенные для семафоров:

- Когда счетчик текущего числа ресурсов становится больше 0, семафор переходит в свободное состояние;
- Если этот счетчик равен 0, то семафор занят;
- Система не присваивает отрицательных значений счетчику текущего числа ресурсов;
- Счетчик текущего числа ресурсов не может быть больше максимального числа ресурсов.

Объект семафор создается вызовом **`CreateSemaphore`**.

Опишем решение нашей задачи. Надо создать семафор со счетчиком максимального числа ресурсов, равным 5, при этом изначально ни один из ресурсов не доступен. Поскольку счетчику текущего числа ресурсов присвоен 0, семафор находится в занятом состоянии. А значит, любой поток, ждущий семафор просто засыпает.

Поток получает доступ к ресурсу, вызывая одну из **`Wait`**-функций и передавая ей, описатель семафора, который охраняет этот ресурс. Wait-функция проверяет у семафора счетчик текущего числа ресурсов: если его значение больше 0 (семафор свободен), уменьшает значение этого счетчика на 1, и вызывающий поток остается планируемым. Семафоры выполняют операцию проверки и присвоения на уровне атомарного доступа.

Если **`Wait`**-функция определяет, что счетчик текущего числа ресурсов равен 0 (семафор занят), система переводит вызывающий поток в состояние ожидания. Когда другой поток увеличит значение этого счетчика, система вспомнит о ждущем потоке и снова начнет выделять ему процессорное время (когда он захватит ресурс, он уменьшит значение счетчика на 1).

Мьютексы

Объекты ядра ***мьютексы*** гарантируют потокам взаимоисключающий доступ к единственному ресурсу. Кроме стандартного заголовка, они содержат счетчик рекурсии и переменную, в которой запоминается идентификатор потока. Мьютексы ведут себя точно так же как и критические секции. Кроме того, единственный объект мьютекс позволяет синхронизировать доступ к ресурсу нескольких потоков из разных процессов; при этом можно задать максимальное время ожидания для доступа к ресурсу.

Идентификатор потока определяет, какой поток захватил мьютекс, а счетчик рекурсий - сколько раз. Как правило, с помощью мьютексов защищают блок памяти, к которому обращаются множество потоков. Мьютексы гарантируют, что любой поток получит монопольный доступ к блоку памяти, тем самым обеспечивают целостность данных.

Для мьютексов определены следующие правила:

- Если идентификатор его потока равен 0, то мьютекс не захвачен ни одним из потоков и находится в свободном состоянии;
- Если идентификатор его потока не равен 0, то мьютекс захвачен одним из потоков и находится в занятом состоянии;

Создается объект мьютекс вызовом **`CreateMutex`**.

ЛАБОРАТОРНАЯ РАБОТА 4

СИНХРОНИЗАЦИЯ

Теория

Многопоточность внутри одного процесса или нескольких основывается на способности процесса координировать свою работу с работой другого процесса. В этой лабораторной работе мы рассмотрим синхронизацию потоков внутри адресного пространства одного процесса (а дальше - синхронизацию различных процессов).

Все потоки процесса выполняются в его адресном пространстве и используют общие ресурсы для решения общей проблемы.

Синхронизирующий механизм позволяет нескольким потокам координировать работу. Существует два вида синхронизации: синхронная и асинхронная.

Пусть два потока 1 и 2 выполняются в одном процессе:

при синхронном сценарии (*Рисунок 4.5*) поток 1 выполняет часть работы, затем поток 2 выполняет вторую часть работы, которая основана на результатах работы 1-ой части. А затем поток 1 выполняет 3-ю часть работы, которая требует завершения 2-ой части.

Этот сценарий требует, чтобы потоки были в состоянии точно координировать исполнение определенных частей работы.

В асинхронном сценарии (*Рисунок 4.6*) - потоки 1 и 2 работают с тремя частями: поток 1 заканчивает первую часть работы, а затем поток 2 заканчивает

2-ю часть параллельно с потоком 1, который будет заканчивать 3-ю часть.

Две части (2 и 3) выполняются одновременно, если в системе не один процессор.

Рисунок 4.5 Синхронные операции

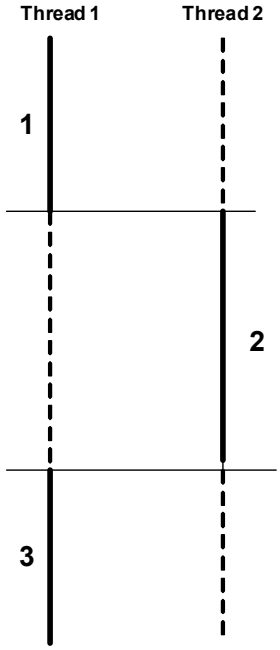
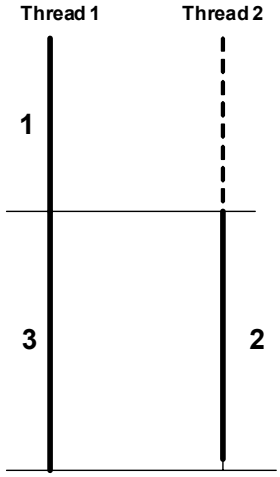


Рисунок 4.6 Асинхронные операции



Потоки 1 и 2 предполагается выполнять параллельно в одном адресном пространстве. Все переменные, описанные в программе процесса, являются разделяемыми (кроме автоматических, которые определяются в стеке потока). Переменные сегмента данных - статические переменные в С программе - разделяются (используются совместно) потоками процесса. Если два или более потока получают доступ к разделенным переменным, выполняя критическую секцию (и по крайней мере хотя бы один задает значение переменной) - будут

проблемы. Решение проблемы - использовать механизм на уровне ОС для взаимного исключения. Если один поток находится в критической секции, тогда другие потоки не входят в критическую секцию до тех пор, пока первый там находится.

Для синхронизации потоков для координации и взаимного исключения выполняется мы будем использовать синхронизирующие объекты (***Executive Obfects***):

- Event - событие
- Mutex - мутант
- Semaphore - семафор
- Timer - таймер.

Поведение всех этих синхронизационных объектов аналогично семафорам Dijkstra:

```
P(s): [while (s = = 0) wait( ); s-- ;]
V(s): [s++;]
```

Здесь P и V - атомарные операции.

`Wait` позволяет другим потокам выполняться пока s=0. Каждый ***объект NT*** (***dispather object***), который строится на базе объекта ядра, содержит переменную состояния, которая позволяет объекту находиться в состоянии ***свободен*** (***signaled***) или ***занят*** (***nonsignaled***). Например, когда процесс или поток заканчивается, его ***объект*** (***dispather object***) переходит в ***свободное*** (***signaled***) состояние. Различные операции приводят к изменению состояния объекта. Программно можно проверять состояние объекта, используя функции ожидания ***WaitForSingleObject*** и **WaitForMultipleObject**.

Процесс и поток - эти объекты создаются в занятом состоянии и остаются в нем всю свою жизнь. Когда процесс или поток заканчиваются, используя **`ExitProcess`** или **`ExitThread`**, их состояние меняется, они переходят из занятого состояния в свободное.

```
ThrdHandle = CreateThread(….);

WaitForSingleObject(thrdHandle, INFINITE);
CloseHandle(thrdHandle);
```

Когда поток создался, он начинает существовать в занятом состоянии. Когда поток заканчивается, он меняет свое состояние на свободное. Тем временем контролирующий процесс блокирует себя используя **`WaitForSingleObject`** до тех пор, пока не закончится поток (child).

Схема при исполнении таймера такая же. Он создается в занятом состоянии. Поток ждет истечения промежутка времени, используя **WaitForSingleObject** с описателем таймера до тех пор, пока он не изменит состояние на свободное.

Функции ожидания

Функции ожидания **WaitForSingleObject** и используются с любыми объектами синхронизации.

```
DWORD WaitForSingleObject(
  HANDLE hObject,
  DWORD dwMilliseconds);
```

Когда поток вызывает эту функцию, первый параметр, **hObject**, идентифицирует объект, который может находиться в состоянии занят/свободен. Второй параметр, **dwMilliseconds**, указывает, сколько времени (в миллисекундах) поток готов ждать освобождения объекта. Если мы задаем **INFINITE**, то это значит, что вызывающий поток готов ждать этого события хоть целую вечность. Правда, передача **INFINITE** не всегда безопасна. Если объект так и не перейдет в свободное состояние, то вызывающий поток никогда не проснется; правда процессорное время он при этом тратить не будет.

```
WaitForSingleObject(hProcess, INFINITE);

Рассмотрим еще один пример:

DWORD dw = WaitForSingleObject( hProcess, 5000);
switch( dw )
{
  case WAIT_OBJECT_0:
// процесс завершается
break;
            case WAIT_TIMEOUT:
    // процесс не завершился в течение 5000 мс
    break;
  case WAIT_FAILED:
    // неправильный вызов функции
    break;
}
```

Вызывающий поток не получит процессорное время, пока не завершится указанный процесс или не пройдет 5000 мс (смотря что случится раньше).

Значение, возвращаемое функцией **WaitForSingleObject**, указывает, почему вызывающий поток снова стал планируемым. Если объект освободился, то функция вернет **WAIT_OBJECT_0**, а если истекло время

таймаута, то **WAIT_TIMEOUT**. Если передан неверный параметр, функция возвращает - **WAIT_FAILED** (для выяснения конкретной причины вызовите **GetLastError)**.

Функция **WaitForMultipleObject** работает аналогично, но позволяет ждать освобождения сразу нескольких объектов или какого-то одного из заданного списка объектов:

```
DWORD WaitForMultipleObjects(
  DWORD dwCount,
  CONST HANDLE* phObjects,
  BOOL fWaitAll,
DWORD dwMilliseconds);
```

Параметр **dwCount** задает количество объектов ядра (его значение должно быть в пределах от 1 до **MAXIMUM_WAIT_OBJECTS=64**). Параметр **phObjects** - это указатель на массив описателей объектов ядра.

WaitForMultipleObjects приостанавливает поток и заставляет его ждать освобождения либо всех заданных объектов ядра, либо одного из них. Параметр **fWaitAll** и задает этот ваш выбор. Если он равен **TRUE**, то функция не даст потоку возобновить работу пока не освободятся все объекты.

Параметр **dwMilliseconds** идентичен одноименному параметру функции **WaitForSingleObject**.

Возвращаемое функцией значение сообщает, почему возобновилось выполнение вызвавшего ее потока. Если вы задали **fWaitAll TRUE**, то функция вернет **WAIT_OBJECT_0**, если все объекты перешли в свободное состояние. Если же **fWaitAll FALSE**, то функция вернет управление, как только один из объектов(любой) перейдет в свободное состояние. Чтобы узнать какой именно объект освободился, необходимо вычесть за возвращаемого значения **WAIT_OBJECT_0**.

```
HANDLE h[3];
h[0] = hPracess1;
h[1] = hPracess2;
h[2] = hPracess3;
DWORD dw = WaitForMultipleObjects(3, h, FALSE, 5000);
switch( dw )
{
  case WAIT_FAILED:
    // неправильный вызов
    break;

  case WAIT_TIMEOUT:
    // ни один из объектов не освободился в течение 5000 мс
    break;

  case WAIT_OBJECT_0 + 0:
```

```
     // завершился процесс h[0]
     break;

case WAIT_OBJECT_O + 1:
     // завершился процесс h[1]
     break;

case WAIT_OBJECT_O + 2:
     // завершился процесс h[2]
     break;
}

#define  N….
.............
HANDLE    thrdHandle[N;]
…….
for(i=0; i<N;i++)
{
thrdHandle[i] = Create Thread(…);
}
……
WaitForMultipleObjects(N, thrdHandle,TRUE,INFINITE);
```

Что случится если **dwMilliseconds=0** при вызове **WaitForSingleObject** или **WaitForMultipleObjects?** (т.е. таймаут немедленно истекает при вызове).

Объекты

Объекты события используется для того, чтобы сообщить о том, что что-то произошло в данном потоке, другим потокам.

```
HANDDE CreateEvent(
  PSECURITY_ATTRIBUTES psa,
  BOOL fManualReset,
  BOOL fInitialState,
PCTSTR pszName);
```

Параметр **fManualReset** (булева переменная) сообщает системе, какого типа событие вы хотите создать:

- **TRUE** - со сбросом вручную;
- **FALSE** - с авто сбросом.

Параметр **fInitialState** определяет начальное состояние события:

- **TRUE** - свободное;
- **FALSE** - занятое

После того как система создает объект событие, **CreateEvent** возвращает описатель события, который помещается в таблицу описателей данного процесса. Ненужный больше объект событие

следует, как всегда закрывать, используя **CloseHandle.** Создав событие, вы можете управлять его состоянием, используя **SetEvent** и **ResetEvent**.

```
BOOL SetEvent(HANDLE hEvent);
BOOL ResetEvent(HANDLE hEvent);
```

Для событий с авто сбросом действует определенное правило. Когда ожидающий его поток наконец то его дождался, объект событие автоматически сбрасывается в занятое состояние (отсюда и произошло название).

Объекты события используется для того, чтобы сообщить о том, что что-то произошло в данном потоке, другим потокам. Можно использовать ***auto-reset*** или **manual-reset**. Повторим, что **auto-reset** позволяет только одному потоку понять, что объект событие перешел в состояние свободен, а затем он немедленно переходит обратно в состояние занят. Использование **manual-reset** дает понять многим потокам, что событие произошло, но объекты события должны быть перегружены используя **SetEvent**.

Используем события для реализации сценария, представленного на рисунке (*Рисунок 4.5*).

```
int main ( )
{
// this is threadX
……..
evntHandle[ PART2 ] = CreateEvent( NULL, TRUE, FALSE, NULL );
evntHandle[ PART3 ] = CreateEvent( NULL, TRUE, FALSE, NULL );
……..
CreateThread(…, threadY,…);
// Do part 1
……..
SetEvent( evntHandle[ PART2 ] );
WaitForSingleObject( evntHandle[ PART3 ], INFINITE );
ResetEvent( evntHandle[ PART3 ] );
// Do part 3
……..
}

DWORD WINAPI threadY ( LPVOID )
{
……..
WaitForSingleObject( evntHandle[ PART2 ], INFINITE );
ResetEvent( evntHandle[ PART2 ] );
// Do part 2
……..
SetEvent( evntHandle[ PART3 ] );
……..
}
```

Мьютексы

Объект ***мьютекс*** **(*mutex*)** также существует исключительно для синхронизации, хотя он специально был создан для управления критическими секциями. Объект мьютекс может принадлежать потоку владельцу или он может быть без владельца. Владение означает, что поток держит мьютекс. Так если мьютекс в состоянии занят, то владеющий поток в критической секции, защищаемой им.

Поток может стать владельцем мьютекса при его создании, получив его описатель (*open*) или используя функцию ожидания. Он может освободить мьютекс (перевести его в состояние свободен), используя функцию **ReleaseMutex**.

```
HANDLE CreateMutex(
LPSECURITY ATTRIBUTES lpMutexAttributes,
// указать на атрибут безопасности
BOOL bInitialOwner, // флаг начального собственника
LPCTSTR IpName     // указатель на имя
     );
```

Атрибут **bInitialOwne***r* определяет будет ли вызывающий поток владельцем мьютекса.

Если **bInitialOwner = TRUE** (и вызов функции успешен), мьютекс будет создан в состоянии занят и вызывающий поток будет его владельцем. Подобно другим функциям для создания объектов **CreateMutex** вернет ошибку, если использовать имя, которое уже используется (**GetLastError** вернет **ERROR ALREADY EXISTS**).

После того, как мьютекс создан, любой поток процесса может его использовать. Если поток не является владельцем мьютекса, но хочет им стать, он должен использовать функцию ожидания. Подобно функции ожидания для объектов других типов возврат из функции ожидания произойдет, если объект перейдет в состояние свободен. Успешное ожидание (вы можете проверить код возврата) позволяет вызываемому потоку стать владельцем, а состояние объекта изменится на занятое. **ReleaseMutex** освобождает мьютекс и его состояние меняется на свободное.

Объекты мьютексы можно использовать для решения проблем критических секций. Предположим потоки ***X*** и ***Y*** разделяют ресурс ***R.*** Оба потока выполняют вычисления, обращаются к ***R*** и затем выполняют вычисления снова. Так как ***R*** разделяемый ресурс доступ к нему это критическая секция.

```
int main( )
{
//This is controlling thread
```

```
……..
// Open resource R
//Create the Mutex Object with no owner  ( signaled )
……Create Thread(…, workerThrd,…); // Create thread X
……Create Thread(…, workerThrd,…); // Create thread Y
…….
}

DWORD WINAPI workerThrd( LPVOID )
{

…….
while(….)
{
// выполнить работу
…….
//получить mutex
while (WaitForSingleObject(mutexR) != WAIT_OBJECT_0);
//Доступ к ресурсу R
ReleaseMutex(mutexR);
}

.....

}
```

Семафоры

```
HANDLE CreateSemaphore(
   PSECURITY_ATTRIBUTES psa,
   LONG lInitialCount,
   LONG lMaximumCount,
   PCTSTR pszName);
```

Параметр **lMaximumCount** сообщает системе максимальное число ресурсов. Параметр lInitialCount указывает, сколько из этих ресурсов доступно изначально.

Объект имеет внутреннюю переменную, принимающую значение от **0** до **lMaximumCount** (должен быть > 0). Когда объект создается, начальное значение внутренней переменной может быть установлено любым в этом промежутке и задается **lInitialCount**. Состояние объекта семафора определяется значением внутренней переменной.

```
HANDLE hSem = CreateSemaphore(NULL, 0, 5, NULL);
```

Внутреннее значение объекта семафора может изменять, используя функции. Функция ожидания уменьшает значение внутренней переменной. Функция **ReleaseSemaphore** увеличивает значение внутренней переменной (счетчика).

```
BOOL ReleaseSemaphore (
HANDLE hSemafhore, //описатель объекта семафора
LONG lReleaseCount, //число + к текущему значению
```

```
LPLONG lPreviousCount //указатель на предыдущие значения
);
```

lReleaseCount - число, которое прибавляется к значению семафора. **lPreviousCount** - указатель на переменную, которая должна содержать значение, которое было до вызова **ReleaseSemaphore** (может быть **NULL**, если вы не заботитесь о предыдущем значении).

Когда использовать семафоры?

Предположим потоки **X** и **Y** оба используют ресурс **R** (который состоит из кусочков), каждый может потребовать **K**- кусочков и использовать их некоторое время, а затем вернуть.

```
#define N……
int main()
{

//это - контролирующий поток
…….
//создать объект семафор
SemaphoreR = CreateSemaphore( NULL, 0, N, NULL );
…..CreateThread(…..,WorkerThrd,…);  //создать поток X
…..CreateThread(…..,WorkerThrd,…);  //создать поток Y
......
}

DWORD WINAPI WorkerThrd(LPVO1D)
{
while(….)
{
//выполнить некоторую работу
…..
for (i=0; i<k; i++
while( WaitForSingleObject(Semaphore) != WAIT_OBJECT_O )
……
//освободить к - кусочков
Release Semaphore (SemaphoreR,K,NULL);
.........
```

Таймеры

```
HANDLE CreateWaitableTimer(
   PSECURITY_ATTRIBUTES psa,
   BOOL fManualReset,
   PCTSTR pszName);
```

Параметр **fManualReset** задает тип ожидаемого таймера: со сбросом вручную или с автосбросом. Когда освобождается таймер со взбросом вручную, возобновляется выполнение всех ожидающих его потоков, а когда в свободное состояние переходит таймер с авто сбросом - лишь одного из потоков.

Объект ожидаемый таймер всегда создается в занятом состоянии. Чтобы сообщить таймеру, в какой момент он должен перейти в свободное состояние, необходимо вызвать функцию **SetWaitableTimer** (см. MSDN).

Критические секции

```
const int MAX_TIMES = 1000;
int g_nIndex = 0;
DWORD g_dwTimes[MAX_TIMES];
DWORD WINAPI  FirstThread(PVOID pvParam)
{
   while(g_nIndex < MAX_TIMES)
   {
g_dwTimes[g_nIndex] = GetTickCount();
g_nIndex++;
   }

   return 0;
}

DWORD WINAPI  SecondThread(PVOID pvParam)
{
   while(g_nIndex < MAX_TIMES)
   {
g_nIndex++;
g_dwTimes[g_nIndex - 1] = GetTickCount();
   }
   return 0;
}
```

Если бы обе функции выполнялись независимо, то каждая заполняла бы массив **g_dwTimes** набором возрастающих чисел. Но в нашем случае обе функции будут одновременно обращаться к глобальному массиву **g_dwTimes**. Допустим пусть первым начал выполняться поток, выполняющий **SecondThread**, и значение счетчика **g_nIndex** увеличилось до 1. А дальше система вытеснила ее поток и начала исполнять **FirstThread**, а та заносит в **g_dwTimes[1]** системное время. После этого процессор вновь переключается на выполнение второго потока и уже тот присваивает элементу **g_dwTimes[1-1]** значение системного времени. Так как эта операция выполняется позже, то значение системного времени, записанное в 0-ой элемент массива, будет больше чем в первый.

Теперь попробуем исправить этот фрагмент, используя критическую секцию.

```
const int MAX_TIMES = 1000;
int g_nIndex = 0;
DWORD g_dwTimes[MAX_TIMES];
CRITICAL_SECTION g_cs;
```

```
DWORD WINAPI  FirstThread(PVOID pvParam)
{
  for( BOOL fContinue = TRUE; fContinue; )
  {
    EnterCriticalSection(&g_cs);
if (g_nIndex < MAX_TIMES)
    {
g_dwTimes[g_nIndex] = GetTickCount();
g_nIndex++;
    }
    else fContinue = FALSE;
    LeaveCriticalSection(&g_cs);
  }
  return 0;
}


DWORD WINAPI  SecondThread(PVOID pvParam)
{
  for( BOOL fContinue = TRUE; fContinue; )
  {
    EnterCriticalSection(&g_cs);
if (g_nIndex < MAX_TIMES)
    {
g_nIndex++;
g_dwTimes[g_nIndex-1] = GetTickCount();
    }
    else fContinue = FALSE;
    LeaveCriticalSection(&g_cs);
  }
  return 0;
```

Необходимо было создать экземпляр структуры данных **CRITICAL_SECTION - g_cs,** а затем вставить вызов **EnterCriticalSection** перед началом работы с разделяемыми ресурсами (у нас это **g_nIndex и g_dwTimes**) и вызов **LeaveCriticalSection** после окончания работы с ними.

Если у вас есть ресурсы, которыми разные потоки пользуются вместе, вы можете поместить их в критическую секцию и она будет их защищать. Вызов **EnterCriticalSection** позволяет выяснить свободна или занята критическая секция. **EnterCriticalSection** допускает вход потока в критическую секцию, если определяет, что та свободна. В противном случае (критическая секция занята) эта функция заставит поток ждать пока она не освободится.

Закончив работу с критическим ресурсом, поток должен вызвать функцию **LeaveCriticalSection**. Таким образом, он уведомляет систему о том, что критическая секция освободилась. Если вы забудите это сделать, то система будет считать, что ресурс все еще занят и не позволит обратиться к нему другим ждущим потокам.

Элементы структуры **CRITICAL_SECTION** необходимо инициализировать до обращения какого-либо потока к защищенному ресурсу. Структура инициализируется вызовом:

```
VOID InitializeCriticalSection(PCRITICAL_SECTION pcs);
```

InitializeCriticalSection должна быть вызвана до того, как один из потоков обратится к **EnterCriticalSection**. Если критическая секция никому больше не нужна, удалите ее вызвав:

```
VOID DeleteCriticalSection(PCRITICAL_SECTION pcs);
```

Нельзя удалять критическую секцию, если она еще нужна какому-нибудь потоку.

Чему нужно научиться?

Необходимо научиться синхронизировать работу потоков процесса

Задание

Уровень 1 (A)

В этой лабораторной вы должны создать два потока, которые выполняются в одном адресном пространстве (в одном процессе):

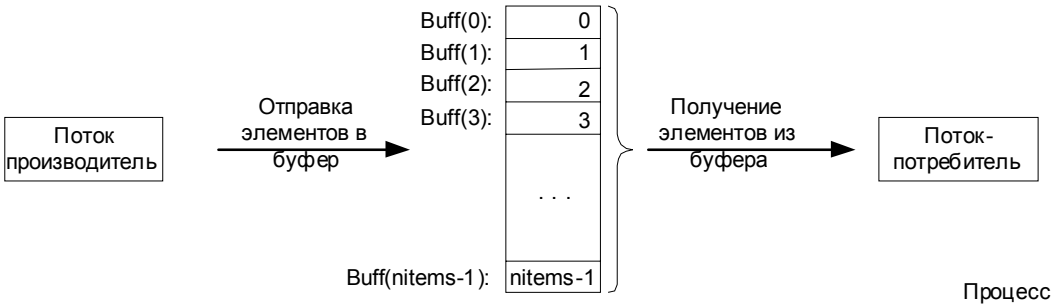
- поток - производитель;
- и поток - потребитель.

Целочисленный массив **buf** содержит производимые и потребляемые данные (данные совместного использования). Производитель просто устанавливает значение **buf[0]=0**, **buf[1]=1** и т.д.

Код должен удовлетворять трем требованиям:

- потребитель не должен пытаться извлечь значение из буфера, если буфер пуст;
- производитель не должен пытаться поместить значение в буфер, если буфер полон;
- состояние буфера должно описываться общими переменными (индексами, счетчиками, указателями связанных списков и т.д. - см. *Рисунок 4.7*).

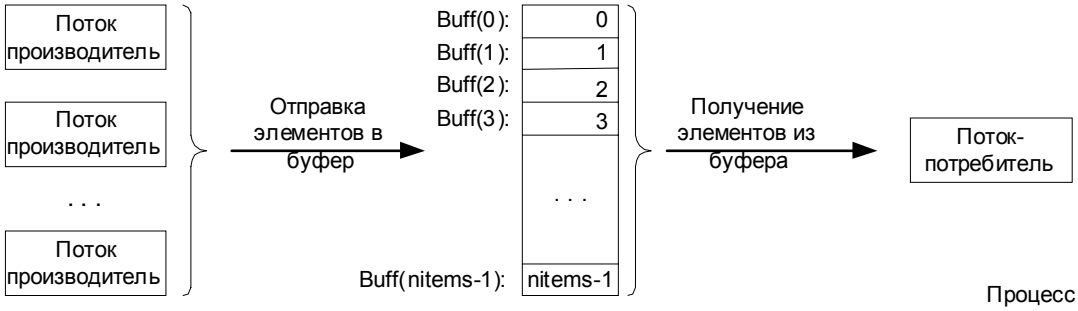
Рисунок 4.7 Производитель и потребитель



Уровень 2 (А)

Пусть имеется несколько производителей и один потребитель (См. *Рисунок 4.8*).

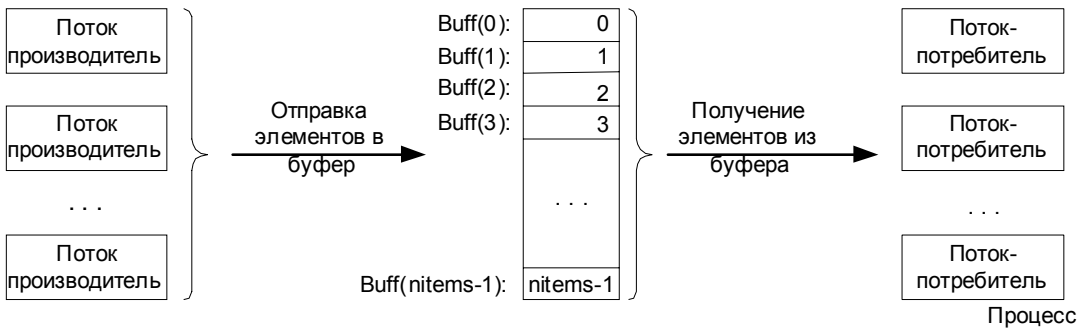
Рисунок 4.8 Производители и потребитель



Уровень 3 (А)

Пусть имеется несколько производителей и несколько потребителей (См. *Рисунок 4.9*).

Рисунок 4.9 Производители и потребители



Скелет кода

```
#define N…..
//разделяемые переменные, включая семафоры
Semaphore mutex = 1;
Semaphore full = 0;
Semaphore empty = N;
buf_type buffer[ N ];

int main( )
{
…….
//создать производителя и потребителя
….. CreateThread(…..., Producer,….);
….. CreateThread(…..., Consumer,…);
……
}

Producer( )
{
buf_type *next, *here;
while( TRUE )
{
produce_item( next );
P( empty );
P( mutex );
here = obtain(empty);
V( mutex );
copy_buffer( next, here );
P( mutex );
release(here,fullPool);
V( mutex );
V( full );
}
}

Comsumer( )
{
buf_type *next, *here;
while( TRUE )
{
produce_item( next );
P( full );
P( mutex );
here = obtait( full );
V( mutex );
copy_buffer(next, here);
P( mutex );
release( here, emptyPool );
V(mutex);
V(empty);
}
}
```

Глава

5

Средства IPC

Процессы должны иметь возможность общаться и разделять между собой общие ресурсы и данные. Для того чтобы сделать это возможным, ОС должна поддерживать определенные механизмы. Такие механизмы получили название ***межпроцессного взаимодействия*** (***interprocess communications, IPC***).

Рассмотрим следующие средства IPC:

- ***Совместное использование*** объектов несколькими процессами для синхронизации. Существует три механизма, позволяющие процессам совместно использовать одни и те же объекты:
- ***Каналы*** и ***сокеты*** предназначены для передачи данных между процессами, выполняющимися на одной машине и на разных машинах в сети.
- ***Секции*** и ***проекции*** позволяют процессам использовать одни и те же блоки данных.

Совместное использование объектов

Бывает необходимо разделять объекты между потоками, исполняемыми в разных процессах.

Описатели объектов имеют смысл только в конкретном процессе, поэтому разделение объектов между несколькими процессами - задача непростая. Главная причина, почему описатели объектов определены для процесса - устойчивость ОС. Если бы описатели были бы общесистемными, то один прцесс мог бы легко получить описатель объекта, используемого другим процессом, и устроить ему "веселую жизнь". Другая причина - защита. Объекты защищены и процесс, прежде чем оперировать с ними, должен получить к ним доступ.

В следующих подразделах мы рассмотрим три механизма, позволяющие процессам совместно использовать одни и те же объекты:

- ***именованные объекты;***
- ***наследование описателей объектов;***
- ***дублирование описателей объектов.***

Именованные объекты

Для того чтобы позволить нескольким процессам совместно использовать одни и те же объекты, можно задать имена для этих объектов. Например, можно создать объекты с именами, используя следующие функции:

```
HANDLE CreateMutex(
   PSECURITY_ATTRIBUTES psa,
   BOOL bInitialOwner,
   PCTSTR pszName);

HANDLE CreateEvent(
   PSECURITY_ATTRIBUTES psa,
   BOOL bManualReset,
   BOOL bInitialState,
   PCTSTR pszName);

HANDLE CreateSemaphore(
   PSECURITY_ATTRIBUTES psa,
   LONG lInitialCount,
   LONG lMaximumCount,
   PCTSTR pszName);

HANDLE CreateWaitableTimer(
   PSECURITY_ATTRIBUTES psa,
   BOOL bManualReset,
   PCTSTR pszName);
```

Последний параметр **pszName** у всех этих функций одинаковый. Передавая в нем *NULL*, вы создаете безымянный объект. И вы можете разделять этот объект между процессами либо через наследование, либо через дублирования описателей (см. следующие разделы Наследование описателей объектов и Дублирование описателей объектов). А чтобы разделить объект по имени, вы должны присвоить ему какое-нибудь имя. Тогда вместо **NULL** в параметре **pszName** нужно передать адрес строки с именем, завершаемым нулевым символом. Имя может быть длиной до **MAX_PATH** (260 символов). Правила именования не определены, поэтому, создавая объект с именем (например, **TvkObj**), вы не застрахованы оттого, что в системе нет объекта с таким именем.

```
HANDLE hMutex = CreateMutex(NULL, FALSE, "TvkObj");
HANDLE hSem = CreateSemaphore(NULL, 1, 1, "TvkObj");
DWORD dwErrorCode = GetLastError();
```

После выполнения этого фрагмента **dwErrorCode** будет равен **ERROR_INVALID_HANDLE** (6).

Теперь рассмотрим, как разделять объекты между процессами по именам. Пусть процесс **A** вызывает функцию:

```
HANDLE hMutexProcessA = CreateMutex(NULL, FALSE, "TvkObj");
```

В этом случае система создает новый объект мьютекс и присваивает ему имя **TvkObj**. Спустя некоторое время другой процесс, например **C**, порождает процесс **B**, который выполняет следующий код:

```
HANDLE hMutexProcessB = CreateMutex(NULL, FALSE, "TvkObj");
```

При вызове система сначала проверяет, не существует ли уже объект с таким именем. И если существует, то ядро проверяет тип этого объекта. Поскольку мы пытаемся создать объект того же типа, то система проверяет права доступа вызывающего процесса к этому объекту. Если все права есть, то в таблице описателей процесса **B** создается новая запись, указывающая на существующий объект. Если вызывающий объект не имеет соответствующих прав доступа или типы объектов с одинаковыми именами не совпадают, то возвращается **NULL**.

Однако, хотя процесс **B** удачно вызывает **CreateMutex**, новый объект мьютекс он не создает. Вместо этого он получает свой собственный описатель уже существующего объекта мьютекса. Счетчик открытых описателей объекта увеличивается на 1, и теперь этот объект не разрушится, пока его описатели не закроют оба процесса - **A** и **B**. Значения описателей объекта мьютекса в обоих процессах, скорее всего, будут разными.

Внимание!!! Вызывая **CreateMutex**, процесс B передает атрибуты защиты и второй параметр. Если объект с указанным именем уже существует, то эти параметы игнорируются. Можно и нужно определять создается ли новый объект или открывает уже существующий, используя **GetLastError**.

```
HANDLE hMutex = CreateMutex(&sa, FALSE, "TvkObj");
if (GetLastError() ==ERROR_ALREADY_EXISTS)
{
  // открыт описатель существующего объекта
}
else
{
  // создан совершенно новый объект
}
```

Вместо вызова **Create**-функции процесс может обратиться к одной из следующих **Open**-функций:

```
HANDLE OpenMutex(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);

HANDLE OpenEvent(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);

HANDLE OpenSemaphore(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);

HANDLE OpenWaitableTimer(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);

HANDLE OpenFileMapping(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);
```

Все эти функции имеют один прототип. Последний параметр определяет имя объекта и в нем нельзя передать **NULL**, только адрес строки с нулевым символом в конце.

Эти функции просматривают единое пространство имен объектов, пытаясь найти совпадение:

- если объекта ядра с указанным именем нет, то функция возвращает **NULL**, а **GetLastError** - **ERROR_FILE_NOT_FOUND**;
- если объект с заданным именем существует, то проверяется его тип:
  - если тип соответствует, указанному вами типу, то система проверяет, разрешен ли запрашиваемый доступ (**dwDesiredAccess**);
  - если такой вид доступа разрешен, таблица описателей в вызывающем процессе обновляется, и счетчик открытых описателей объекта возрастает на 1.;

Главное отличие между вызовом **Create** и **Open** в том, что при отсутствии указанного объекта функция **Create** создает его, а функция **Open** уведомляет об ошибке.

Наследование описателей объектов

Наследование применимо, только когда процессы связаны родственными отношениями (*parent - child*). Пусть у родительского процесса есть один или несколько описателей объектов, и он хочет, создвая дочерний процесс, передать ему их по наследству. Для того чтобы это получилось, родительский процесс должен:

- при создании объектов сообщить системе, что нужны наследуемые описатели данных объектов;
- при создании родительским процессом дочернего процесса с помощью **CreateProcess** параметр **bInheritHandles** должен быть **TRUE**.

Чтобы создать наследуемый описатель, родительский процесс выделяет и инициализирует структуру **SECURITY_ATTRIBUTES**, а затем передает ее адрес функции **Create<Class>**.

```
SECURITY_ATTRIBUTES sa;
sa.nLength = sizeof(sa);
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE;    // делаем возвращаемый описатель наследуемым
HANDLE hMutex = CreateMutex(&sa, FALSE, NULL);
…
```

Это код инициализирует структуру **SECURITY_ATTRIBUTES** таким образом, чтобы был создан объект с защитой по умолчанию и возвращаемый описатель был бы наследуемым.

А теперь рассмотрим флаги, которые хранятся в таблице описателей объектов процесса. В каждой записи таблицы присутствует флаг, указывающий является ли данный описатель наследуемым или нет. Если вы, создавая объект, передаете в параметре **PSECURITY_ATTRIBUTES** значение **NULL**, то получите ненаследуемый описатель и этот флаг будет **0**. А если элемент **bInheritHandle** равен **TRUE**, то флагу присваивается **1**.

Таблица 5.1 Таблица объектов процесса с двумя действительными записями

Индекс	*Указатель на блок памяти объекта*	*Маска доступа*	*Флаги*
1	**0xF0000000**	0x????????	**0x00000000**
2	0x00000000	неприменим	неприменим
3	**0xF0000010**	0x????????	**0x00000001**

По таблице можно сказать, что данный процесс имеет доступ к двум объектам, причем описатель 3 - ***наследуемый***, а описатель 1 - ***ненаследуемый*** (*Таблица 5.1*).

Рассмотрим второй этап - родительский процесс пораждает дочерний, используя функцию ***CreateProcess***. Подробно эту функцию мы уже рассматривали, а сейчас нас интересует только параметр **`bInheritHandles`**. Передавая в этом параметре **FALSE**, мы сообщаем системе, что дочерний процесс не должен *наследовать* описатели, которые помечены как *наследуемые* в таблице описателей родительского процесса. Если передать **TRUE**, то дочерний процесс наследует описатели, которые помечены как *наследуемые* в таблице описателей родительского процесса. В этом случае ОС, создав дочерний процесс, не дает ему немедленно начать работу. Она сначала создает в нем новую (пустую) таблицу описателей. Затем считывает таблицу описателей родительского процесса и копирует все ее *наследуемые* записи в таблицу описателей дочернего процесса - причем в те же позиции. Это означает, что описатели будут одинаковые в родительском и дочернем процессе и это очень важно.

Кроме того, система увеличивает значения счетчиков открытых описателей соответствующих объектов, так как эти объекты теперь используются обоими процессами. Чтобы уничтожить какой-то объект, его описатель должны закрыть оба процесса. Если сразу после возврата управления функцией **`CreateProcess`** родительский процесс закроет свой описатель объекта, это не помешает дочернему процессу работать с ним.

Далее показана таблица (*Таблица 5.2*) описателей дочернего процесса перед началом его исполнения. Записи 1 и 2 не инициализированы. Однако индекс **3** идентифицирует объект по тому же (что и в родительском) адресу **0xF0000010.** Маска доступа и флаги в родительском и дочернем процессах идентичны. Если дочерний процесс породит новый процесс (внука по отношению к родительскому процессу), то внук также унаследует даный описатель объекта, а счетчик открытых описателей этого объекта вновь увеличится на 1.

Таблица 5.2 Таблица объектов дочернего процесса (после того как он унаследовал от родительского один наследуемый описатель)

Индекс	Указатель на блок памяти объекта ядра	Маска доступа	Флаги
1	0x00000000	неприменим	неприменим
2	0x00000000	неприменим	неприменим
3	**0xF0000010**	0x????????	**0x00000001**

Наследуются только описатели объектов, которые существуют на момент создания дочернего процесса. Если родительский процесс создаст после этого новые объекты и пометит их как наследуемые, то они будут недоступны дочернему процессу.

Дочерний процесс не знает, что он унаследовал от родителя какие-то описатели. Поэтому в дочерний процесс необходимо, каким то образом передать значение наследуемого описателя объекта (например, как аргумент командной строки). Тогда дочерний процесс, имея значение описателя, сможет получить доступ к унаследованному объекту. Все так хорошо получается только потому, что значение описателя общего объекта в родительском и дочернем процессах одинаково.

Дублирование описателей объектов

Для совместного использования объектов несколькими процессами можно использовать функцию **DuplicateHandle:**

```
BOOL DuplicateHandle(
  HANDLE hSourceProcessHandle,
  HANDLE hSourceHandle,
  HANDLE hTargetProcessHandle,
  HANDLE hTargetHandle,
  DWORD dwDesiredAccess,
  BOOL bInheritHandle, // флаг наследования
  DWORD dwOptions);
```

Рассмотрим пример. Пусть **S** (Source) - это процесс источник, имеющий доступ к какому-нибудь объекту, **T** (Target) - процесс приемник, который хочет получить доступ к тому же объекту, а **D** - процесс, который будет вызывать функцию **DuplicateHandle**.

Таблица описателей процесса **D** содержит два индекса 1 и 2 (*Таблица 5.3*). Первый описатель идентифицирует объект процесс **S,** а второй объект процесс **T.**

Таблица 5.3 Таблица описателей процесса **D**

Индекс	*Указатель на блок памяти объекта ядра*	*Маска доступа*	*Флаги*
1	**0xF0000000** ***объект процесс S***	0x????????	0x00000000
2	**0xF0000010** ***объект процесс T***	0x????????	0x00000000

Теперь рассмотрим таблицу описателей прцесса **S** (*Таблица 5.4*), содержащую единственную запись со значением описателя равным **2.** Этот описатель может идентифицировать объект любого типа, но только не процесс.

Таблица 5.4 Таблица описателей процесса **S**

Индекс	*Указатель на блок памяти объекта ядра*	*Маска доступа*	*Флаги*
1	0x00000000	неприменим	неприменим
2	**0xF0000020** ***объект любого типа***	0x????????	0x00000000

В следующей таблице (*Таблица 5.5*) показано, что именно содержится в таблице описателей процесса **T** перед вызовом процессом **D** функции **`DuplicateHandle`**. В ней присутствует всего одна запись со значением описателя равным **2**, а запись с индексом 1 пока пуста.

Таблица 5.5 Таблица описателей процесса **T** перед вызовом **`DuplicateHandle`**

Индекс	*Указатель на блок памяти объекта ядра*	*Маска доступа*	*Флаги*
1	0x00000000	неприменим	неприменим
2	**0xF0000030** ***объект любого типа***	0x????????	0x00000000

Пусть **D** теперь вызовет **`DuplicateHandle`**:

```
DuplicateHandle(1, 2, 2, &hObj, 0, TRUE, DUPLICATE_SAME_ACCESS);
```

После вызова таблица описателей процесса **T** изменится (*Таблица 5.6*).

Таблица 5.6 Таблица описателей процесса T после вызова **`DuplicateHandle`**

Индекс	*Указатель на блок памяти объекта ядра*	*Маска доступа*	*Флаги*
1	0xF0000020	0x????????	0x00000001
2	0xF0000030 объект любого типа	0x????????	0x00000000

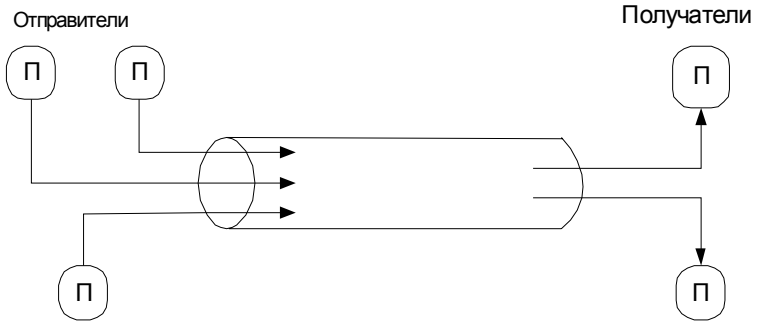
Вторая строка таблицы описателей процесса **S** скопирована в первую строку таблицы описателей процесса **T**.

Каналы (Pipes)

В традиционных реализациях UNIX ***программные каналы*** (***pipe***) представляют собой однонаправленный неструктурированный поток данных фиксированного максимального размера, работающий по принципу FIFO ("первым вошел, первым вышел"). Каналы популярны в UNIX как первичный механизм IPC. Из-за этой популярности, они были включены в IPC механизмы Windows.

Отправители добавляют данные в конец канала, получатели извлекают их из его начала (*Рисунок 5.1*). После того как данные будут прочитаны, они сразу же удаляются из канала и больше недоступны другим получателям. Процесс, осуществляющий попытку чтения из пустого канала, будет приостановлен до тех пор, пока в канале не появятся какие либо данные. Точно так же, если процесс попытается записать в заполненный программный канал, то он будет приостановлен до того момента, пока иной процесс не прочтет данные из канала, тем самым, освободив его.

Рисунок 5.1 Использование канала



Для создания программного канала надо использовать **`CreatePipe`**. Каждый процесс, в праве как читать, так и записывать информацию, а так же производить оба действия сразу (*Рисунок 5.1*). Однако в большинстве случаев программный канал используется двумя процессами, на двух его концах. Операции ввода-вывода над каналами аналогичны операциям над файлами. Для этого можно использовать **`ReadFile`** / **`WriteFile`**. Процесс иногда может и не знать о том, что работает с каналом, а не с обычным файлом.

С точки зрения межпроцессорного взаимодействия программные каналы являются эффективным способом передачи данных от одного процесса другому. Однако они обладают несколькими ограничениями:

- так как чтение данных из канала приводит к их удалению, то он не может быть использован для широковещательной передачи информации нескольким адресатам;
- данные канала интерпретируются как поток байтов, границы сообщения заранее не известны. Если отправитель посылает через канал несколько объектов различной длины, получатель не имеет возможности определять, как много объектов ему было передано он, так же не знает, где заканчивается один объект и начинается другой;
- когда данные из канала считываются несколькими процессами, отправитель не в состоянии передать данные какому-то определенному получателю. Точно так же при наличии нескольких отправителей не существует способа определения, какой из них отослал данные.

Сокеты

Эталонная модель OSI

Задача сетевого программного обеспечения (ПО) состоит в приеме запроса (обычно ввод-вывод) от приложения на одной машине, передаче его на другую, выполнение запроса на удаленной машине и возврате результата на первую машину. В ходе этих операций запрос неоднократно трансформируется. Высокоуровневый запрос вроде "считать x байт из файла y на машине z" требует, чтобы ПО определило, как достичь машины z, и какой коммуникационный протокол она понимает. Затем запрос должен быть преобразован для передачи по сети - например, разбит на короткие пакеты данных. Когда запрос достигнет другой стороны, нужно проверить его целостность, декодировать и послать соответствующему компоненту ОС. По окончании обработки запрос должен быть закодирован для обратной передачи по сети.

Чтобы помочь поставщикам в стандартизации и интеграции их сетевого ПО, международная организация по стандартизации (ISO) определила программную модель пересылки сообщений между компьютерами. Эта модель получила название ***эталонной модели OSI*** (***Open Systems Interconnection***). В ней определено семь уровней ПО (*Рисунок 5.2*).

Рисунок 5.2 Модель OSI

Эталонная модель OSI - идеал, точно реализованный лишь в очень немногих системах, но часто используемый при объяснении основных принципов работы сети. Каждый уровень на одной машине считает, что он взаимодействует с тем же уровнем на другой машине. На данном уровне обе машины "разговаривают" на одном языке, или протоколе. Но в действительности сетевой запрос должен сначала пройти до самого нижнего уровня на первой машине, затем он передается по несущей среде и уже на второй машине вновь поднимается до уровня, который его поймет и обработает.

Уровни OSI

Задача каждого уровня в том, чтобы предоставлять сервисы более высоким уровням и скрывать от них конкретную реализацию этих сервисов. Краткое описание каждого сетевого уровня:

- ***Прикладной уровень (application layer).*** Обрабатывает передачу данных меду двумя сетевыми приложениями, включая проверку прав доступа, идентификацию взаимодействующих машин и инициацию обмена данными;

- ***Презентационный уровень*** (***presentation layer***). Отвечает за форматирование данных, в том числе решает, должны ли строки заканчиваться парой символов "возврат каретки/перевод строки" (CR/LF) или только символом "возврат каретки" (CR), надо ли сжимать данные, кодировать и т. д.;

- ***Сеансовый уровень*** (***session layer***). Управляет соединением взаимодействующих приложений, включая высокоуровневую синхронизацию и контроль за тем, какое из них "говорит", а какое "слушает";

- ***Транспортный уровень*** (***transport layer***). На передающей стороне разбивает сообщения на пакеты и присваивает им порядковые номера, гарантирующие прием пакетов в должном порядке. Кроме того, изолирует сеансовый уровень от влияния изменений в составе оборудования;

- ***Сетевой уровень*** (n***etwork layer***). Создает заголовки пакетов, отвечает за маршрутизации, контроль трафика и взаимодействие с межсетевой средой. Это самый высокий из уровней, который понимает топологию сетей, т. е. физическую конфигурацию машин в них, ограничения пропускной способности этих сетей и т. д.;

- ***Канальный уровень*** (***data-link layer***). Пересылает низкоуровневые кадры данных, ждет подтверждений об их приеме и повторяет передачу кадров, потерянных в ненадежных линиях связи;

- ***Физический уровень*** (***physical layer***). Передает биты по сетевому кабелю или другой физической несущей среде.

Пунктирными линиями (*Рисунок 5.2*) показаны протоколы, применяемые для передачи запроса на удаленную машину. Как уже говорилось, каждый сетевой уровень считает, что он взаимодействует с эквивалентным уровнем на другой машине, который использует тот же протокол. Набор протоколов, передающих запросы по сетевым уровням, называется ***стеком протоколов***.

Windows Sockets (Winsock)

Windows Sockets (***Winsock***)- это Microsoft-реализация BSD (*Berkeley Software Distribution)* Sockets, программного интерфейса, с 80-х годов ставшего стандартом, на основе которого UNIX-системы взаимодействовали через Интернет. Поддержка сокетов в Windows существенно упрощает перенос сетевых приложений из UNIX в Windows. Winsock включает большую часть функциональности BSD Sockets, а так же содержит специфические расширения от Microsoft, развитие которых еще продолжается. Winsock поддерживает как надежные коммуникации, ориентированные на *логические соединения,* так и на ненадежные коммуникации, не требующие логических соединений.

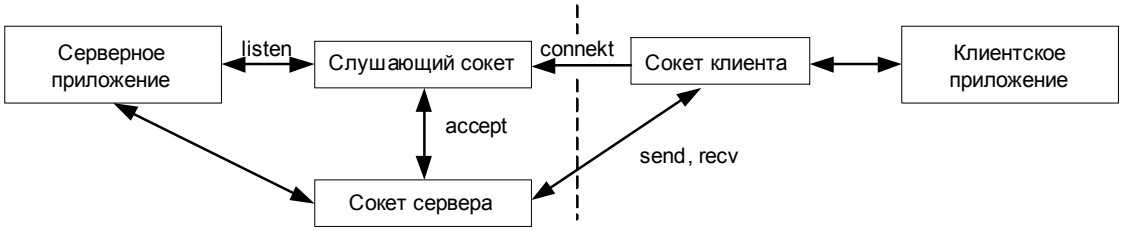
После инициализации Winsock *API* вызовом инициализирующей функции первый шаг Winsock-приложения - создание сокета, представляющего конечную точку коммуникационного соединения. Сокеты должны быть *привязаны* к адресу локального компьютера, поэтому вторым шагом приложения является привязка сокета. Winsock не зависит от конкретного протокола, поэтому можно выбрать любой

протокол (*NetBEUI, TCP/IP, IPX*), установленный в системе, где работает *Winsock*. По окончании привязки дальнейшие действия клиента и сервера расходятся - равно как они различны и для сокетов разных типов (ориентированных и не ориентированных на логические соединения). *Winsock*- сервер, ориентированный на логические соединения, выполняет на сокете операцию **`listen`**, указывая число соединений, которое он может поддерживать на этом сокете.

Далее он выполняет операцию **`accept`**, чтобы клиент мог подключиться к сокету. При наличии ждущего запроса на соединение вызов **`accept`** завершается немедленно. В ином случае он завершается лишь после поступления запроса на соединение. После того как соединение установлено, функция **`accept`** возвращает новый сокет, представляющий серверную сторону соединения. Сервер может выполнять операции приема и передачи данных с помощью функций **`recv`** и **`send`**.

Клиенты, ориентированные на логические соединения, подключаются к серверу вызовом Winsock-функции **`connect`** с указанием адреса удаленного компьютера. После того как соединение установлено, клиент может посылать и принимать сообщения через свой сокет. Рисунок (*Рисунок 5.3*) иллюстрирует коммуникационную связь между клиентом и сервером.

Рисунок 5.3 Связь между клиентом и сервером



После привязки к адресу сервер, не требующий логических соединений, ничем не отличается от аналогичного клиента: он посылает и получает сообщения через сокет, просто указывая удаленный адрес для каждого сообщения. При использовании сообщений, не требующих логических соединений (т. е. дейтаграмм), отправитель получает код ошибки в ходе следующей операции приема, если предыдущее сообщение не было доставлено.

ЛАБОРАТОРНАЯ РАБОТА 5.1

СОВМЕСТНОЕ ИСПОЛЬЗОВАНИЕ ОБЪЕКТОВ

Теория

См. *“Совместное использование объектов” на стр. 145*.

Чему нужно научиться?

Нужно научиться:

- синхронизировать работу потоков процесса (**A**);
- синхронизировать работу процессов с использованием именованного объекта (**B**);
- синхронизировать работу процессов с использованием наследования описателей объектов (**C**);
- синхронизировать работу процессов с использованием дублирования описателей объектов (**D**).

Задания A, B и C

Уровень 1 (A)

Использовать любые синхронизационные объекты для синхронизации нескольких (например, двух) потоков в одном процессе. В качестве разделяемого ресурса использовать стандартный ввод/вывод. Каждый из потоков должен осуществлять прием строк от пользователя и отвечать пользователю (общаться с пользователем). Причем возможность общения должна быть предоставлена только одному потоку. Передать возможность общения, другому потоку нужно вводя **`next`**, а завершать поток, вводя **`exit`**. Основной (базовый) поток процесса должен закончить работу только в том случае, когда пользователь завершит работу всех вспомогательных потоков.

Уровень 2, 3 (A)

Придумать любую задачу на синхронизацию потоков процесса и реализовать ее.

Уровень 1 (B)

Использовать любой именованный синхронизационный объект для синхронизации нескольких процессов. В нашем случае это будут несколько (два) запущенных экземпляров одного приложения. В качестве разделяемого ресурса использовать стандартный ввод/вывод. Только один из запущенных экземпляров приложения (процесс) получает возможность принимать строки от пользователя и отвечать пользователю (общаться с пользователем). Передать возможность общения другому запущенному экземпляру приложения (процессу) нужно вводя **`next`**, а завершить процесс, вводя **`exit`**.

Уровень 2, 3 (B)

Придумать любую задачу на синхронизацию процессов с использованием именованного объекта и реализовать ее.

Уровень 1 (C)

Синхронизировать родительское и дочернее приложения (два родственных процесса). Процесс-родитель создает любой синхронизационный объект и передает его описатель по наследству создаваемому процессу-ребенку (описатель передать как аргумент командной строки). В качестве разделяемого ресурса использовать стандартный ввод/вывод. Только один из родственных процессов получает возможность принимать строки от пользователя и отвечать пользователю (общаться с пользователем). Передать возможность общения, другому процессу нужно вводя **`next`**, а завершить процесс, вводя **`exit`**.

Уровень 2 (C)

Придумать любую задачу на синхронизацию родственных процессов с использованием наследования описателей и реализовать ее.

Уровень 3 (C)

Синхронизировать два процесса. Один процесс создает любой синхронизационный объект и дублирует его описатель для другого процесса. В качестве разделяемого ресурса использовать стандартный ввод/вывод. Только один из процессов получает возможность принимать строки от пользователя и отвечать пользователю (общаться с пользователем). Передать возможность общения, другому процессу нужно вводя **`next`**, а завершить процесс, вводя **`exit`**.

Скелет кода для задания А

```
#include <stdio.h>
#include <windows.h>
#include <string.h>
#include <conio.h>

DWORD WINAPI Thread1(LPVOID lpParameter)
{
…printf("Thread1 begins\n");

while(!bExit)
{
// вставить WaitForSingleObject
gets(str);
printf("User input %s\n", str);

while((strcmp( str,"next"))&&(strcmp(str,"exit")))
{
gets(str);
printf("User input %s\n", str);
}

if( !strcmp(str,"exit") )…

printf("Thread1 release control\n");

// использовать SetEvent
} // while

printf( "Thread1 finished\n" );
…….
return 0;
}

DWORD WINAPI Thread2(LPVOID lpParameter)
{
…….
}

int main( int argc, char* argv[] )
{
printf( "I am main! Hello!\n" );

// создать объекты для синхронизации hNextEvent = CreateEvent(…);

CreateThread(…Thread1,….);
…….
return 0;
}
```

Скелет кода для задания B

```
#include <windows.h>
…

HANDLE hEvent;

int main( int argc, char* argv[] )
{
.......
hEvent = CreateEvent(…,"Named_event");
……
}
```

Скелет кода для задания C (Parent)

```
#include <windows.h>
……..
HANDLE hEvent;
SECURITY_ATTRIBUTES SecurityAttributes;
…

int main( int argc, char* argv[] )
{
SecurityAttributes.bInheritHandle = TRUE;
…

// define path to child process
strcpy(path, "d:\\tvk\\child\\debug\\child.exe ");

hEvent = CreateEvent(&SecurityAttributes,…);

itoa((int) hEvent, str, 10);
strcat(path, str);

if(!CreateProcess(…,path,...,TRUE,…))
{
printf( "Could not create child process!\n" );
CloseHandle(hEvent);
return 0;
}

while(TRUE)
{
……..
}
……
}
```

Скелет кода для задания C (Child)

```
#include <windows.h>
.........

HANDLE hEvent;

int main( int argc, char* argv[] )
{
if( argc != 2 )
{
printf( "Invalid parameter!\n" );
return 0;
}

hEvent = ( HANDLE )atoi( argv[1] );

while(TRUE)
{
.........
}
........
}
```

ЛАБОРАТОРНАЯ РАБОТА 5.2

ИСПОЛЬЗОВАНИЕ КАНАЛОВ

Теория

Потоки одного процесса используют одно и тоже адресное пространство, поэтому им легко обмениваться информацией друг с другом. Мы уже научились использовать объекты, для того чтобы поток одного процесса мог сигнализировать потоку другого процесса. Однако если процессы хотят обмениваться данными, то необходимо использовать ***каналы*** (***pipes***) - механизм, который поддерживает передачу данных между адресными пространствами процессов.

Модель файлового ввода/вывода

Мы рассмотрели, как представляются в виде объектов процессы, потоки, таймеры и синхронизационные примитивы. В случае процессов и потоков такое представление служит двум целям: иметь описатель для описания поведения процесса и обеспечить простую модель для программистов манипулирования этим описателем.

Программист может использовать функцию ожидания (синхронизации), чтобы определить, когда объект перестанет существовать. Подобно процессам и потокам, объект исполнительной системы ***файл*** служит для хранения информации об открытом файле. Этот объект имеет обычные для любого объекта свойства безопасности, наследования и синхронизации.

При создании файла создается объект файл и его описатель помещается в таблицу описателей процесса. Файловые объекты используются для представления других системных ресурсов - устройств. ***File*** - это именованный поток символов с последовательным доступом. Указатель файла (64 бита) ассоциирован с каждым экземпляром открытого файла. Когда файл открывается, его указатель равен 0. Когда мы читаем или записываем К-байт в файл, указатель перемещается на К.

Файловые объекты (**`CreateFile`**) используются для представления файлов, каталогов, параллельных и последовательных портов, каналов, сокетов и консоли. Указатель файла можно также перемещать вручную, используя **`SetFilePointer`**.

```
DWORD SetFilePointer(
HANDLE hfile, // описатель файла
LONG lDistanceToMove,
// число байт для перемещения указателя файла (low 32 bits)
PLONG lpDistanceToMoveHigh,
// число байт для перемещения указателя файла (high 32 bits)
DWORD dwMoveMethod // как перемещать
);
```

Использование 64-битового указателя файла вызывает некоторые трудности при программировании. **LdistanceToMove** - это 32 битовое число со знаком, на которое перемещается указатель файла вперед или назад от позиции, определенной в **dwMoveMethod** (от начала файла, от конца файла или от текущего положения). **LpDistanceToMoveHigh** - это старшие биты 64-битового значения.

Необходимо заметить, что **SetFilePointer** манипулирует указателем файла в описателе файла (а не в объекте файле), поэтому все потоки изменяющие положение в файле, используя один описатель файла будут разделять один указатель. Таким образом, если один поток вызвал **SetFilePointer**, а затем выполнил **ReadFile**, мы не можем гарантировать, что другой не выполнит другую операцию **SetFilePointer** между первой операцией и чтением.

```
HANDLE CreateFile(
DWORD dwDesiredAccess, // режим доступа
DWORD dwShareMode, // режим совместного доступа
LPSECURITY_ATTRIBUTES lpSecurityAttributes,
// атрибуты безопасности
DWORD dwCreationDesposition, // как создать файл
DWORD dwFlagsAndAttributes, // атрибуты файла
HANDLE hTemplate
// описатель файла с атрибутами для копирования
);
```

LpFileName задает имя создаваемого или открываемого файла. **DwDesireAccess** этот параметр может принимать только три значения:

- **0** (***no access*** - нет доступа)
- **GENERIC_READ** (чтение)
- **GENERIC_WRITE** (запись).

Можно использовать их комбинации (**OR**). **DwShareMode** может принимать следующие значения:

- **zero** - использовать совместно нельзя;
- **FILE_SHARE_DELETE** - операции “открыть” будут успешны только в случае, если они требуют доступ-удаление;

- **FILE_SHARE_READ** - следующая операция “открыть” должна быть для чтения;
- **FILE_SHARE_WRITE** - следующая операция “открыть” должна быть для записи.

Можно использовать их комбинации (**OR**). Атрибуты безопасности трактуются также как у любого объекта. **DwCreationDesposition** определяет, что должно происходить, если в момент создания (вызова **CreateFile**) файл уже существует:

- **CREATE_NEW** - если файл существует, то будет ошибка;
- **CREATE_ALWAYS** - переписать существующий файл;
- **OPEN_EXISTING** - открыть существующий файл.

DwFlagsAndAttributes используется для передачи различных опций File Manager, в простейшем случае можно использовать F**ILE_ATTRIBUTE_NORMAL**. **Htemplate** можно использовать для передачи описателя другого файла, который содержит расширенные атрибуты.

```
BOOL ReadFile(
HANDLE hFile, // описатель файла
LPVOID lpBuffer, // адрес буфера для записи данных
DWORD nNumberOf BytesToRead,
// число байт, которые должны быть прочитаны
LPDWORD lpNumberOfBytesRead,
// число действительно прочитанных байт ( указатель )
LPOVERLAPPED lpOverlapped // адрес структуры данных
);
```

При успешном вызове возвращается не 0 значение. Если возвращен не 0, а число действительно прочитанных байт 0, то это означает, что указатель файла в момент вызова указывал за границу файла.

```
BOOL WriteFile(
HANDLE hFile, // описатель файла
LPVOID lpBuffer,
DWORD nNumberOf BytesToWrite,
LPDWORD lpNumberOfBytesWritten,
LPOVERLAPPED lpOverlapped
);
```

Использование файлового ввода/вывода - каналы (pipes)

Каналы это IPC абстракция для передачи информации из адресного пространства одного процесса в адресное пространство другого. Каналы популярны в UNIX и из-за этого они были включены Windows. Windows поддерживает два варианта каналов:

- анонимные
- и именованные.

Анонимные каналы наполовину дуплексные (связь в одну сторону), а именованные каналы полно дуплексные (двух сторонняя связь). Именованные каналы можно использовать в сети.

Каналы имеют конец для чтения и конец для записи, каждый из которых имеет свой собственный описатель. Когда канал создан, функции **ReadFile** и **WriteFile** можно использовать для чтения и записи двух концов канала.

```
BOOL CreatePipe(
PHANDLE hReadPipe,
PHANDLE hWritePipe,
LPSECURITY_ATTRIBUTES lpPipeAttributes,
DWORD nSize
);
```

Вы должны выделить память для описателей чтения и записи (**hReadPipe** *и* **hWritePipe**), задать атрибуты безопасности и определить, сколько байт надо выделить для канала. Можно задать 0, тогда система выделит размер канала по умолчанию. Если вы попробуете прочитать пустой канал (**ReadFile**), то вызывающий поток будет блокирован, пока данные в канале не появятся. Если вы попробуете писать в полный канал (**WriteFile**), то поток будет блокирован, пока не появится свободное место в канале. Таким образом, канал - это умный буфер.

Преимущество в использовании каналов в том, что их могут использовать много процессов. А проблема в том, что анонимный канал создается одним процессом с описателями для чтения и записи в адресном пространстве этого процесса. Как другой процесс получит эти описатели? Стандартная техника передать описатель от одного процесса к другому: использовать глобальное имя (но анонимные каналы безымянные), наследование или дублирование описателей. Классическое решение проблемы в UNIX - наследование описателей. Родитель создает канал и передает описатели как перенаправление стандартного ввода и вывода (**stdin** и **stdout**) детям.

```
int main(int argc, char *argv[])
{
HANDLE readPipe, writePipe;

SECURITY_ATTRIBUTES pipeSA;

STARTUPINFO scrStartInfo, sinkStartInfo;

PROCESS_INFORMATION scrProcessInfo, sinkProcessInfo;
```

```
…
// Создайте канал и не забудьте про наследование
pipeSA.bInheritHandle=TRUE;

if(!CreatePipe(&readPipe,&writePipe,&pipeSA,0))
{
…
ExitProcess(1);
}
// Создайте процесс для записи в канал и не забудьте про наследование
memset(&scrStartInfo,0,sizeof(STARTUPINFO));
scrStartInfo.cb=sizeof(STARTUPINFO);
scrStartInfo.hStdInput=GetStdHandle(STD_INPUT_HANDLE);
scrStartInfo.hStdOutput=writePipe;
scrStartInfo.hStdError=GetStdHandle(STD_ERROR_HANDLE);
scrStartInfo.dwFlags=STARTF_USESTDHANDLES;

if(!CreateProcess(NULL, "d:…\\wpipe.exe",…))
{
…
ExitProcess(1);
}

// Создайте процесс для чтения из канала и не забудьте про наследование
memset(&sinkStartInfo,0,sizeof(STARTUPINFO));
sinkStartInfo.cb=sizeof(STARTUPINFO);
sinkStartInfo.hStdInput=readPipe;
sinkStartInfo.hStdOutput=GetStdHandle(STD_OUTPUT_HANDLE);
sinkStartInfo.hStdError=GetStdHandle(STD_ERROR_HANDLE);
sinkStartInfo.dwFlags=STARTF_USESTDHANDLES;

if(!CreateProcess( NULL, "d:…\\rpipe.exe",…))
{
…
ExitProcess(1);
}
GoseHandle(readPipe);
CloseHandle(writePipe);
```

Пусть созданные процессы используют любые функции ввода/вывода для чтения и записи **stdin / stdout**.

Чему нужно научиться?

Нужно научиться использовать каналы для передачи информации между родственными процессами.

Задания

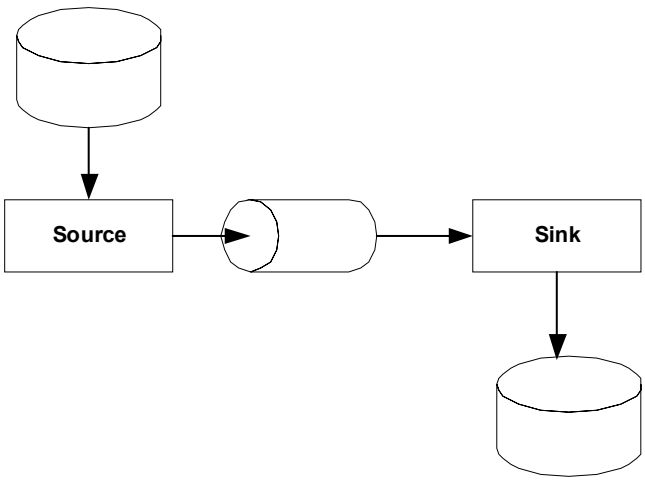
Уровень 1 (A)

У вас должно быть два процесса (*Рисунок 5.4*):

- Первый процесс источник (*source*) - он должен читать информацию из файла и записывать ее в канал;
- Второй процесс приемник (*sink*) - получает информацию из канала и записывает ее во второй файл.

Процессы должны работать с разными скоростями, чтобы протестировать работу.

Рисунок 5.4 Источник и приемник



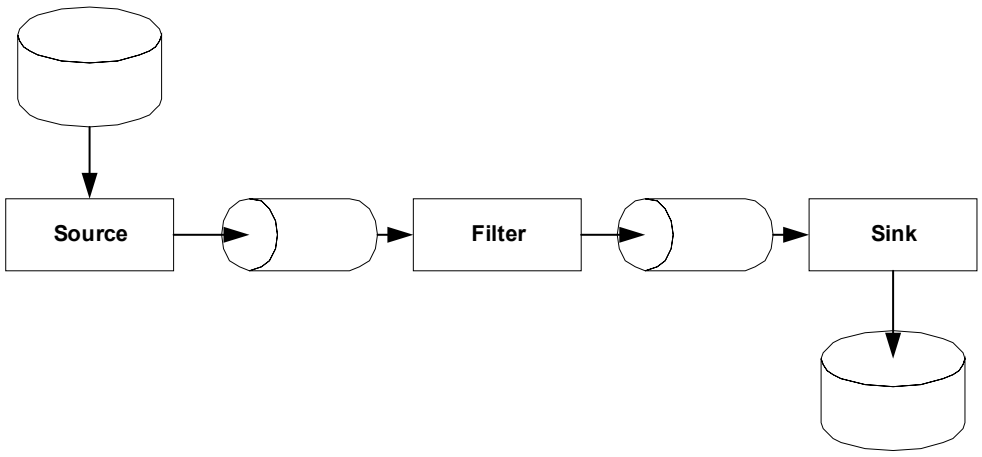
Уровень 2 (A)

У вас должно быть три процесса:

- Первый процесс источник (*source*) - он должен читать информацию из файла и записывать ее в канал;
- Второй процесс фильтр (*filter*) - получает информацию от источника (через канал), выполняет простейшую фильтрацию (например, верхний регистр/нижний регистр) и записывает данные во второй канал. Второй канал используется для связи между фильтром и третьим процессом приемником;
- Третий процесс приемник (*sink*) - получает информацию через канал от фильтра и записывает ее во второй файл.

Процессы должны работать с разными скоростями, чтобы протестировать работу (*Рисунок 5.5*).

Рисунок 5.5 Источник, фильтр и приемник



Уровень 3 (A)

У вас должно быть два процесса:

- клиент
- и сервер.

Необходимо обеспечить следующую функциональность:

- клиент считывает полное имя файла из стандартного потока ввода и записывает его в канал;
- сервер считывает это имя из канала и пытается открыть этот файл для чтения;
  - если попытка успешна, сервер считывает содержимое файла и записывает его в канал;
  - в противном случае сервер возвращает клиенту сообщение об ошибке;
- клиент считывает данные из канала и записывает их в стандартный поток вывода.

Скелет кода

Для того чтобы процессы работали с разными скоростями необходимо вставить некоторую симуляцию в источник, фильтр и приемник.

В Windows существует возможность получения случайных чисел, используя библиотеки.

```
#include <stdlib.p>  //srand() & rand()
#define P_RAND_SEED   1234

int main( int argc, char *argv[] )
{  const int delay = 500;
…..
srand(P_RAND_SEED );
//main loop
while(…){ simulatedWork( rand()%delayFactor );
//Random delay
…..}
……
}
```

ЛАБОРАТОРНАЯ РАБОТА 5.3

ИСПОЛЬЗОВАНИЕ СОКЕТОВ

Теория

Имена

Сетевые имена добавляют дополнительную сложность в IPC в сети. Вы теперь знаете, что для обмена информацией между адресными пространствами на одной машине процессы должны предпринимать усилия. Мы решали эту проблему, используя именованные объекты с именами из глобального пространства имен Windows. Таким образом, если процесс **B** хочет использовать объект, созданный и названный процессом **A,** он должен использовать его имя. Для того чтобы процесс на одной машине мог использовать объекты процесса на другой машине необходимо использовать подобное пространство имен. Только оно должно быть шире, чтобы объединять пространства имен машин в сети. В Internet процесс на машине использует <**net**, **host**, **port**> для идентификации другой машины в сети, а затем использует имя нужного объекта из обычного пространства имен.

Сокеты

Существует между идеей I*nternet* адресов в форме <**net, host, port**> и ***описателями*** (***handles***), которые существуют в адресном пространстве процесса. В BSD версии UNIX была предложена подобная описателям структура данных, названная ***сокеты*** (***socket***), для взаимодействия с сетевыми портами. В BSD ПО для посылки и получения информации было написано так, чтобы оно посылало информацию в *сокет*, далее она передавалась с использованием протокола транспортного уровня и получало, переданную с использованием протокола транспортного уровня информацию из *сокета*. Таким образом, *сокет* это "конец" для TCP.

TCP требует, чтобы связь была установлена перед использованием. После того как оба процесса создали *сокеты*, один из них должен выполнить инициализацию, чтобы его *сокет* выполнял роль виртуального канала. Код на удаленном конце принимает или отвергает требование установить виртуальный канал. Когда пара процессов заканчивает обмен, они должны разорвать установленный виртуальный канал.

Передающий процесс должен определить сетевое имя получателя, но откуда он сможет узнать его *сокет*? Проблема в том, что *сокет* подобен описателю; он создается процессом во время выполнения и существует в его адресном пространстве. Существует возможность для программиста связать *сокет* с именем в сети. Такая процедура называется ***связыванием*** (***binding***) сокета с адресом в Internet.

Пакет WinSock

Пакет Microsoft WinSock использует практически тот же самый код, что и пакет BSD UNIX Socket. Он позволяет программисту использовать UDP и TCP с IP. Мы рассмотрим TCP.

Socket WinSocket создает коммуникационный порт для вызывающего процесса. Socket можно рассматривать как конечную точку для сетевой связи; это индекс в таблице описателей сокетов.

```
SOCKET socket(int af,
int type,
int protocol);
```

Параметр **af** определяет область имен, которая будет использоваться (мы будем использовать **AF_INET**). Параметр **type** определяет природу связи (датаграммы, виртуальные циклы). Мы будем использовать **SOCK_STREAM** для TCP. Если вы хотите использовать UDP и датаграммы, то необходимо задать **SOCK_DGRAM**.

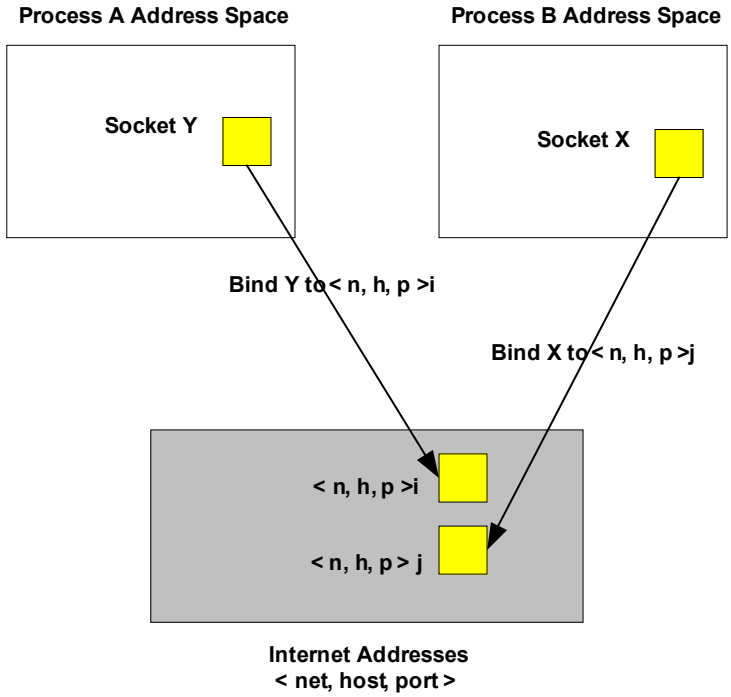
Параметр протокол определяет используемый протокол. Существует несколько протоколов, которые могут быть использованы с заданным типом. Задаваемый протокол должен быть совместим с этим типом. Если вы используете **SOCK_STREAM** и передаете значение 0 для протокола, будет использован TCP, аналогично **SOCK_DGRAM** - UDP. После вызова процесс получит описатель *сокета*.

Для представления сокета в области имен Internet необходимо использовать вызов **bind** (*Рисунок 5.6*).

```
int bind(
SOCKET s,
const  struct sockaddr FAR* name,
int namelen
);
```

Параметр **s** - это значение возвращаемое после вызова **socket** (описатель сокета). Параметр **name** определяет имя сокета в сети <**net, host, port**>, которое будет ассоциироваться с этим сокетом

Рисунок 5.6 Связывание сокета



Если другой поток в Internet пошлет информацию использовав это имя - <**net, host**, **port**>, эту информацию можно прочитать, используя **s.**

```
struct sockaddr {
u_short sa_family;
char sa_data[14];
};
```

Sockaddr можно использовать со всеми доменами адресов; **sockaddr_in** используется с TCP/IP.

```
struct sockaddr_in {
short sin_family;
u_short sin_port;
struct in_addr sin_addr;
char sin_zero[8];
};
```

Так как **bind** работает с различными типами структур **sockaddr**, необходимо задать ***длину структуры*** используемого типа (***namelen***). Ее необходимо задать для того, чтобы **bind** мог определить длину второго параметра.

Вызов **connect** используется потоком клиента (его сокет **s**) для инициализации соединения (установления предварительного соединения) с сервером (сокетом сервера).

```
int connect(
SOCET s,
const  struct sockaddr FAR* name,
int namelen
);
```

Параметр имя <**net, host, port**> определяет имя *сокета* сервера. Это должно быть тоже имя, которое используется в **bind**. Конечно, существует много причин, из-за которых вызов **connect** может произойти с ошибкой. Например, заданное имя не существует в пространстве имен (для определения причин используйте **WSAGetLastError**). **Namelen** задает длину второго параметра.

Вызов ***listen*** выполняется процессом сервера, который собирается принять вызов ***connect*** для установления соединения (виртуального канала). После этого процесс сервера готов принимать вызовы ***connect*** от сетевых клиентов.

```
int listen(
SOCKET s,
int backlog
);
```

Параметр **backlog** задает максимальное число устанавливаемых соединений, которые процесс будет поддерживать в данное время на этом ***сокете.*** Каждый входящий запрос на установление соединения будет поставлен в очередь, до тех пор, пока процесс сервера не будет в состоянии принять этот запрос на обработку (**accept**). Если вызов **listen** не выполнен, то очередь будет нулевой длины и любой запрос на установление соединения от клиентов будет отвергнут.

При успешном вызове **accept** создается новый сокет и его описатель возвращается вызывающей программе. Концами виртуального канала становятся новый сокет процесса сервера, который выполнил **accept**, и оригинальный *сокет* клиента, который выполнил **connect.**

```
SOCKET accept(
SOCET s,
const  struct sockaddr FAR* addr,
int FAR* addrlen
);
```

Пакет WinSock требует вызвать **WSAStartup** перед его использованием и **WSACleanup** после.

```
int WSAStartup(
WORD wVersionRequested,
LPWSADATA lpWSAData
);
int WSACleanup(void);
```

Задания

Уровень 1 (A)

Организуйте общение между двумя потоками в двух различных процессах. Пусть один поток выступает инициатором (*server*) и посылает другому сообщение (*client*). Получатель должен принять посланное сообщение и ответить. Необходимо, чтобы клиент и сервер успешно общались:

- когда они запущены на одной машине;
- когда они запущены на разных машинах.

Уровень 2, 3 (A)

Сделайте так, чтобы сервер мог обслуживать несколько клиентов. И чтобы каждого клиента обслуживал отдельный поток сервера.

Окружение

Используя пакет WinSock, вы должны использовать библиотеку WinSock 2.0. Для этого необходимо добавить wsock32.lib в список используемых библиотек, а также libc.lib (в настройках Visual C++, а именно: Project -> Settings (Alt + F7) в закладке Link в строке Object/ Library modules необходимо добавить библиотеку ws2_32.lib).

Скелет кода Client

```
#include <winsock2.h>
#include <stdio.h>

int main(int argc, char* argv[])
{
SOCKET clientSock;

sockaddr_in serverAddr;
struct hostent* pHostEnt;
char hostName[64];
int addrLen=sizeof(serverAddr);
int nbytes;
char buff[256];

WORD versionRequested;
WSADATA wsaData;

WORD version = MAKEWORD(2,2);

if(WSAStartup(version,&wsaData))
{
printf("("WSA service failed to initialize with error %d \
n",WSAGetLastError());
```

```
exit(1);
};

/*Создаем сокет*/
clientSock=socket(AF_INET,SOCK_STREAM,0);

if(clientSock<0)
{
printf("Socket initialization failed \n");
exit(1);
};
/* Определяем адрес хоста*/
gethostname(hostName,64);
printf("Client is running on %s \n",hostName);

// We are going to connect with Server!!!
if(argc!=0)
strcpy(hostName,argv[1]);
else
gethostname(hostName,64);

printf("We are going to connect with Server %s \n",hostName);

pHostEnt=gethostbyname(hostName);
if(pHostEnt==NULL)
{
printf("Can't get host by name.");
exit(1);
};

/*Соединение с сервером*/

memcpy(&serverAddr.sin_addr, pHostEnt->h_addr,4);
serverAddr.sin_port= htons(40842);
serverAddr.sin_family=AF_INET;

if(connect(clientSock,(sockaddr*)&serverAddr,addrLen)!=0)
{
printf("Connect error. \n");
closesocket(clientSock);
exit(1);
};

//

closesocket(clientSock);

WSACleanup();
}
```

Скелет кода Server

```
#include <winsock2.h>
#include <stdio.h>

int main(int argc, char* argv[])
```

```
{
SOCKET serverSock,newserverSock;

sockaddr_in serverAddr,clientAddr;
struct hostent* pHostEnt;
char hostName[64];
//sockaddr_in sockAddr;
int addrLen=sizeof(serverAddr);
int nbytes;
char buff[256];

WORD versionRequested;
WSADATA wsaData;

WORD version = MAKEWORD(2,2);

if(WSAStartup(version,&wsaData))
{
printf("WSA service failed to initialize with error %d \
n",WSAGetLastError());
exit(1);
};

/* Создание сокета*/
serverSock=socket(AF_INET,SOCK_STREAM,0);
if(serverSock<0)
{
printf("Socket initialization failed with error %d \n",WSAGetLastError());
exit(1);
};

gethostname(hostName,64);
pHostEnt=gethostbyname(hostName);
if(pHostEnt==NULL)
{
printf("Can't get host by name.");
exit(1);
};
printf("Server is running on %s \n",hostName);

/*Связывание сокета*/

memcpy(&serverAddr.sin_addr, pHostEnt->h_addr,4);
serverAddr.sin_port= htons(40842);
serverAddr.sin_family=AF_INET;

if(bind(serverSock,(sockaddr*)&serverAddr,addrLen)!=0)
{
printf("Bind failed with error %d \n",WSAGetLastError());
closesocket(serverSock);
exit(1);
};

/*Сервер слушает*/

if(listen(serverSock,3))
```

```
{
printf("Listen failed with error %d \n",WSAGetLastError());
closesocket(serverSock);
exit(1);
};

/*Установление соединения*/
newserverSock=accept(serverSock,(sockaddr*)&clientAddr,&addrLen);
if(newserverSock<0)
{
printf("Accept failed with error %d \n",WSAGetLastError());
exit(1);
};

closesocket(newserverSock);

WSACleanup();
```

Глава

6

Виртуальная память

На заре компьютерной эпохи нельзя было выполнить программу, размер которой превышал размер физической памяти компьютера. ***Виртуальная память*** (***virtual memory***) - это централизованная система выгрузки на диск содержимого памяти при переполнении последней. Она позволяет программистам создавать и запускать программы, которые требуют памяти больше, чем есть у компьютера.

Память можно описывать в терминах физической и логической структуры, а также способа, которым ОС транслирует одну структуру в другую.

Физическая память организована как последовательность однобайтовых ячеек. Байты пронумерованы от нуля и до общего размера памяти, доступного в данной конфигурации.

Логическая память или ***виртуальная память*** - это способ представления памяти для программы, и в современных операционных системах она редко совпадает с физической структурой памяти. Обычно системы виртуальной памяти используют либо сегментное, либо линейное представление памяти. В персональных компьютерах на основе процессора Intel, начиная с Intel 8086 и заканчивая Intel 80286, была использована сегментная модель. В этом случае, физическая память разделена на последовательные блоки, называемые сегментами. А адрес состоит из номера сегмента и смещения в нем.

С другой стороны, многие процессоры поддерживают линейную структуру адресов. При такой схеме адреса начинаются с 0 и возрастают до верхней границы адресного пространства.

Виртуальное адресное пространство (***virtual address space***) - это набор адресов памяти, которые могут использовать потоки процесса. Каждый процесс имеет отдельное адресное пространство, которое гораздо больше размера физической памяти. Диапазон физических адресов

компьютера ограничен объемом имеющейся у него памяти (у каждого байта уникальный адрес), тогда как диапазон виртуальных адресов ограничен только количеством битов в адресе. Процессор, имеющий 32-разрядный адрес, является обладателем виртуального адресного пространства в 4 Гб.

Система виртуальной памяти должна выполнять две задачи:

- Транслировать, или отображать (map), некоторое подмножество виртуальных адресов каждого процесса в участки физической памяти. Когда поток производит чтение или запись в своем виртуальном адресном пространстве, система виртуальной памяти перед пересылкой данных определяет по виртуальному адресу соответствующий физический адрес;

- Выгружать на диск часть содержимого памяти, когда она переполняется, т.е. когда потоки, выполняющиеся в системе, пытаются использовать больше памяти, чем доступно физически.

Перемещение данных между памятью и диском было бы очень медленным, если бы диспетчер виртуальной памяти перемещал по одному байту за раз. Поэтому виртуальное адресное пространство разделено на блоки равного размера, которые называются ***страницами*** (***pages***). Соответственно, физическое адресное пространство разделяется на блоки, называемые ***страничными*** *фреймами* (***page frames***), которые используются для хранения страниц. В любой момент времени в памяти находится некоторое множество страниц из виртуального адресного пространства каждого процесса. Страницы, находящиеся в физической памяти и доступные немедленно, называются ***действительными страницами*** (***valid pages***). Страницы, находящиеся на диске (или находящиеся в памяти, но недоступные немедленно), называются ***недействительными страницами*** (***invalid pages***). Наглядно это изображено на рисунках (*Рисунок 6.1* и *Рисунок 6.2*).

При обращении потока по виртуальному адресу, который находится на странице, помеченной как недействительная, процессор генерирует ***страничную ошибку*** (***page fault***). Система виртуальной памяти находит нужную страницу на диске и загружает ее в свободный страничный фрейм физической памяти. Когда остается мало доступных страничных фреймов, система виртуальной памяти выбирает фреймы, которые можно освободить и копирует их содержимое на диск. Этот процесс называется ***подкачкой страниц*** (***paging***).

Рисунок 6.1 Отображение виртуальных страниц в физические страничные фреймы

Обработка страничной ошибки - это дорогая операция, которая требует много тактов процессора. Можно снизить затраты за счет выбора оптимального размера страницы. При большем размере страницы мы загружаем в память большее количество данных и, поэтому страничные ошибки будут возникать реже. Но при слишком большом размере, мы можем загрузить лишние данные. Оптимальный размер - 4Кб.

Управление памятью

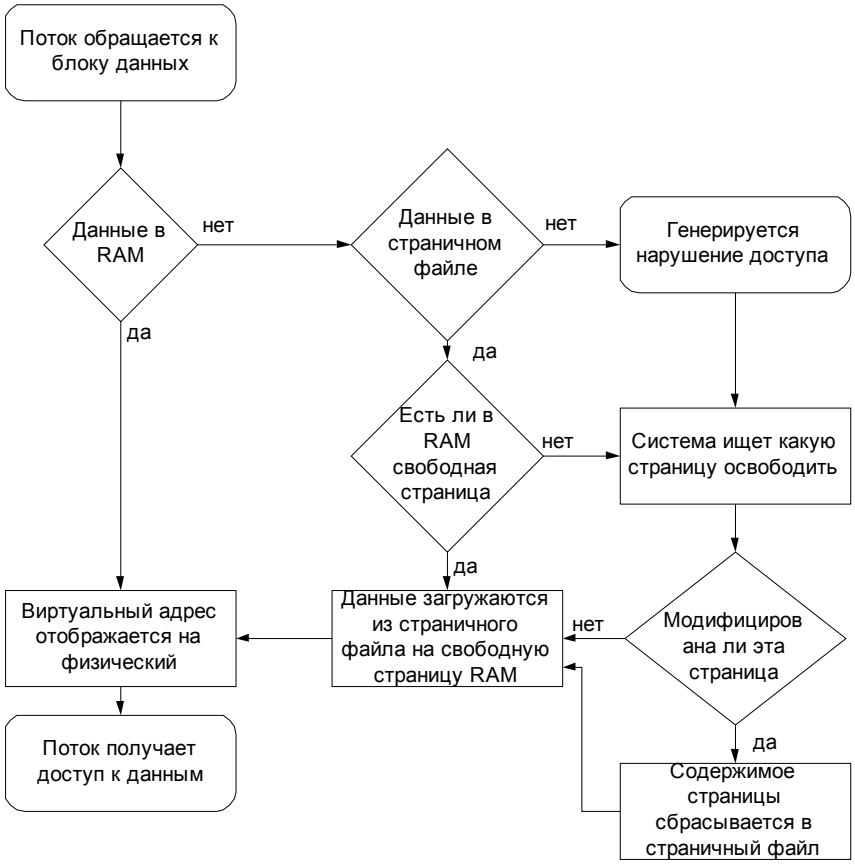
Диспетчер виртуальной памяти предоставляет набор базовых сервисов, при помощи которых процесс может управлять своей виртуальной памятью. Эти сервисы позволяют процессу:

- выделять память в два этапа;
- выполнять чтение/запись из/в виртуальной памяти;
- фиксировать виртуальные страницы в физической памяти;
- получать информацию о виртуальных страницах;
- защищать виртуальные страницы;
- сбрасывать содержимое виртуальных страниц на диск.

Диспетчер виртуальной памяти выделяет память в два этапа:

- ***резервирование;***
- ***передача.***

Рисунок 6.2 Трансляция виртуального адреса в физический



Зарезервированная память (***reserved memory***) - это набор виртуальных адресов, которые диспетчер виртуальной памяти зарезервировал для использования процессом. Это быстрая и дешевая операция. ***Переданной памятью*** (***committed memory***) называется память, для которой диспетчер виртуальной памяти выделил место в ***файле подкачки*** (***paging file***) - файле на диске, в который он записывает виртуальные страницы, когда их надо удалить из памяти. Поток может либо сразу зарезервировать и передать виртуальную память, либо вначале лишь зарезервировать ее, а передавать по мере необходимости.

Резервирование памяти полезно при создании динамических структур данных. Поток резервирует последовательные виртуальные адреса, а передает их, когда по ним необходимо поместить данные.

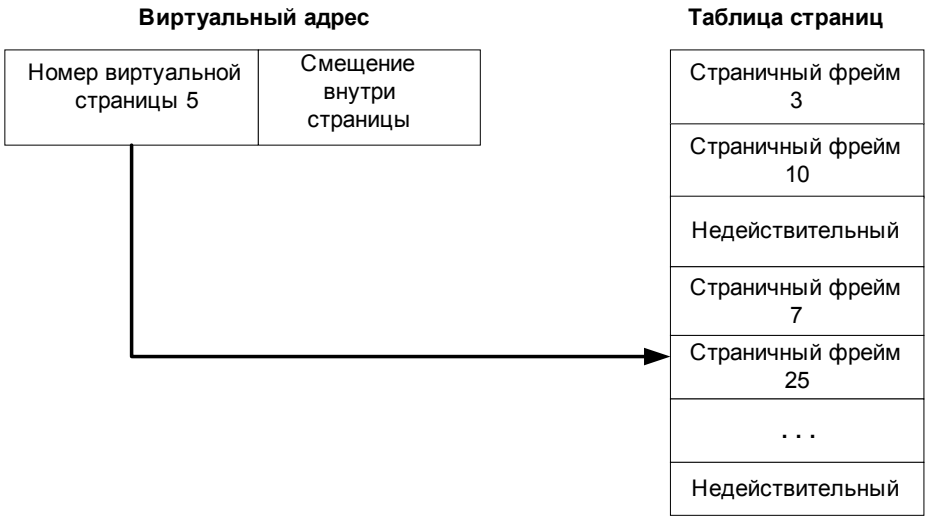
Диспетчер виртуальной памяти списывает с процесса часть его квоты в файле подкачки при передаче, но не при резервировании памяти. Так поток может зарезервировать большой регион виртуальной памяти, но не расходовать квоту до тех пор, пока эта память действительно не понадобится.

Трансляция адресов

Рассмотрим, как Windows увязывает виртуальные адресные пространства процессов со страницами физической памяти.

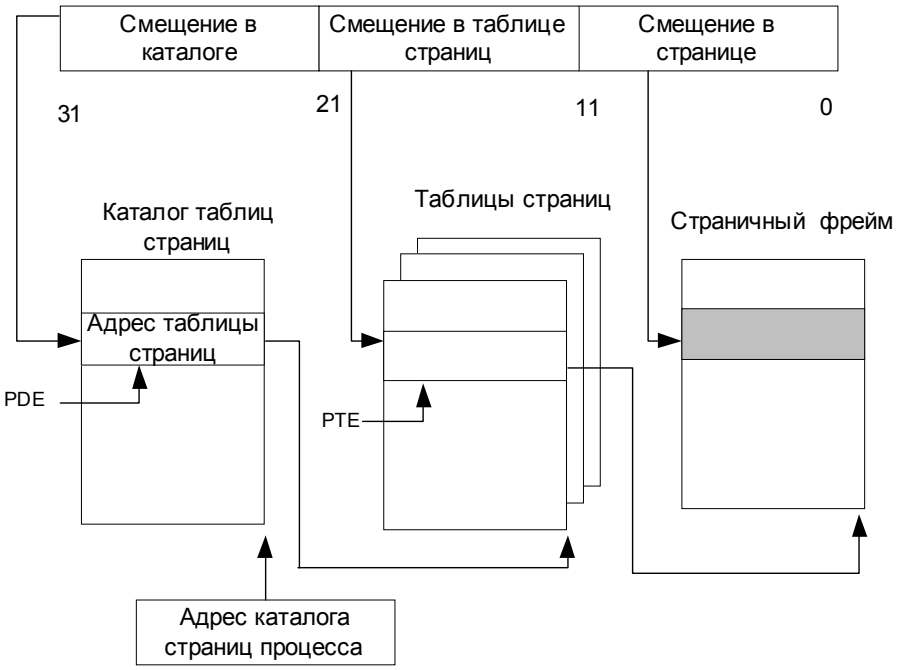
Приложения используют 32-разрядные виртуальные адреса. С помощью структур данных создаваемых и поддерживаемых диспетчером памяти, происходит трансляция этих виртуальных адресов в физические. Каждый виртуальный адрес сопоставляется со структурой системного пространства, которая называется элементом ***таблицы страниц*** (***page table entry***, ***PTE***). Она и содержит физический адрес, соответствующий виртуальному (*Рисунок 6.3*).

Рисунок 6.3 Концептуальная схема таблицы страниц



Для трансляции виртуальных адресов в физические используется двухуровневая таблица страниц. Виртуальный 32-разрядный адрес состоит из трех элементов: ***индекса каталога страниц***, ***индекса таблицы страниц***, и ***индекса байта*** (*Рисунок 6.4*).

Рисунок 6.4 Структура таблицы страниц



Индекс каталога страниц (***page directory index***) применяется для поиска таблицы страниц, содержащий PTE для данного виртуального адреса. С помощью ***индекса таблицы страниц*** (***page table index***) осуществляется поиск PTE, который содержит физический адрес, по которому проецируется виртуальная страница. ***Индекс байта*** (***byte index***) позволяет найти конкретный адрес на физической странице. У каждого процесса есть один ***каталог страниц*** (***page directory***), который представляет собой страницу с адресами всех таблиц страниц для данного процесса.

При трансляции виртуального адреса происходит следующее:

- Определяется адрес каталога страниц текущего процесса. При переключении контекста процесса операционная система заносит этот адрес в специальный регистр процессора;
- Индекс каталога страниц используется для поиска ***элемента каталога страниц*** (***page directory entry, PDE***). Он указывает на ту таблицу страниц, которая нужна для трансляции виртуального адреса. PDE содержит ***номер фрема страницы*** (***page frame number, PFN***) таблицы страниц (если она находится в памяти, таблица может быть выгружена в страничный файл);
- Индекс таблицы страниц используется как указатель для поиска PTE, который определяет, где находится страница. Если она действительная, то PTE содержит PFN соответствующей страницы

физической памяти. Если страница оказывается недействительной, то подсистема управления памятью пытается найти ее и сделать действительной;

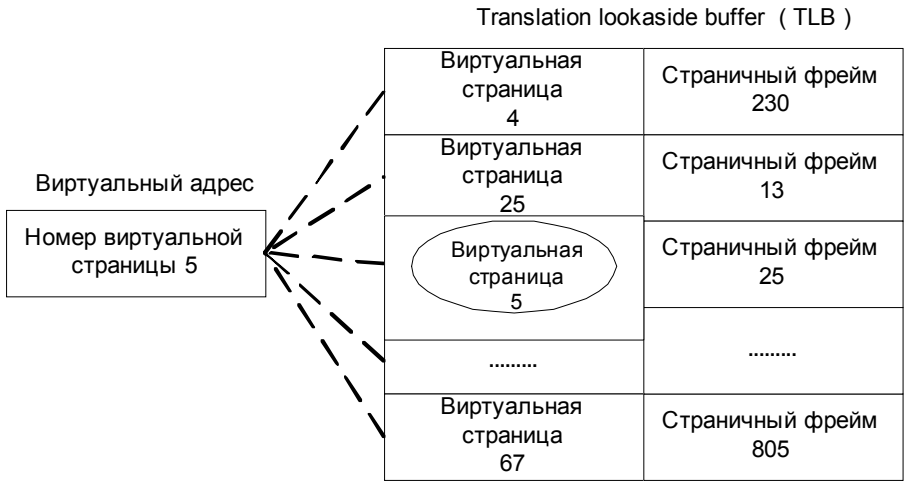
- Если PTE указывает на действительную страницу, для поиска нужных данных используется индекс байта.

Ассоциативный буфер трансляции

Трансляция каждого адреса требует двух операций поиска: сначала нужно найти подходящую таблицу страниц в каталоге страниц, а затем элемент в этой таблице. Выполнение этих операций при каждом обращении по виртуальному адресу снижает быстродействие системы. Поэтому большинство процессоров кэшируют транслируемые адреса и необходимость в повторной трансляции тех же адресов из-за этого отпадает. Процессор типа x86 поддерживает такой кэш в виде массива ассоциативной памяти, называемого ***ассоциативным буфером трансляции*** (***translation lookaside buffer***, ***TLB***).

Ассоциативная память - это вектор, составляющие которого могут одновременно считываться и сравниваться с целевым значением. В случае TLB вектор содержит отображение виртуальных адресов в физические для страниц, которые использовались самыми последними. Упрощенная схема TLB показана на рисунке (*Рисунок 6.5*).

Рисунок 6.5 Ассоциативный буфер трансляции



Для адресов, которые используются часто, с высокой вероятностью будут присутствовать входы в TLB, а значит, трансляция этих виртуальных адресов в физические будет происходить очень быстро. Когда виртуальная страница откачивается на диск, диспетчер виртуальной памяти делает недействительным соответствующий вход

TLB. При обращении процесса к такой странице происходит страничная ошибка, и диспетчер виртуальной памяти вновь переносит страницу в память и снова создает для нее вход TLB.

Для поиска страниц, отсутствующих в TLB, диспетчер виртуальной памяти использует таблицы страниц, создаваемые программно. Концептуально таблица страниц представлена на рисунке (*Рисунок 6.3*).

Вход таблицы страниц (***page table entry, PTE***) содержит всю информацию, необходимую системе виртуальной памяти для поиска страницы, когда поток обращается по адресу в ней. Недействительный вход означает, что страница отсутствует в физической памяти и ее нужно загрузить с диска. Генерируется страничная ошибка, программное обеспечение подкачки страниц загружает запрашиваемую страницу в память и обновляет таблицу страниц. Команда, вызвавшая страничную ошибку, выполняется повторно. Теперь вход таблицы страниц является действительным, и обращение к памяти выполняется успешно.

При использовании 32-разрядных адресов для каждого процесса будет доступно 2^{32} (4Гб). Виртуальные адреса объединяются в страницы по 2^{12} (4Кб), таким образом, в каждом адресном пространстве получается 2^{20} страниц. Если один вход таблицы страниц имеет размер 2^2 (4б), то для отображения всей виртуальной памяти процесса потребуется $(2^{20}*2^2)/2^{12})=1024$ страничных фрейма. И это только для адресного пространства одного процесса.

Стратегия подкачки и рабочие наборы

Обычно система виртуальной памяти определяет три типа стратегии: ***стратегия считывания*** (***fetch policy***), ***стратегия размещения*** (***placement policy***) и ***стратегия замещения*** (***replacement policy***).

Стратегия считывания (***fetch policy***) определяет, когда надо перемещать страницу с диска в память. Можно пытаться загрузить страницы, которые потребуются процессу, до того как он их запросит (заранее). А можно использовать ***стратегию подкачки по запросу*** (***demand paging policy***), в этом случае страница загружается в память только тогда, происходит страничная ошибка.

Диспетчер виртуальной памяти Windows использует алгоритм подкачки по запросу с кластаризацией. Когда возникает страничная ошибка, диспетчер виртуальной памяти загружает страницу, вызвавшую ошибку, вместе с небольшим количеством окружающих ее страниц. Это позволяет минимизировать количество страничных ошибок.

При возникновении страничной ошибки система виртуальной памяти должна определить, в какое место физической памяти следует загрузить эту виртуальную страницу. Здесь начинает действовать ***стратегия размещения*** (***placement policy***). В ОС Windows используется линейная архитектура памяти, поэтому если память не заполнена, диспетчер виртуальной памяти просто выбирает первый страничный фрейм из списка свободных страничных фреймов. Если этот список пуст, то диспетчер просматривает другие списки в заданном порядке (см. *Рисунок 6.7*).

Если страничная ошибка происходит, когда вся физическая память заполнена, то применяется ***стратегия замещения*** (***replacement policy***). Она определяет, какую страницу нужно извлечь из памяти, чтобы освободить место для новой страницы. Обычно используют следующие стратегии:

- ***замещение используемого меньше всего*** (***least recently used***)
- и ***первым пришел, первым ушел*** (***first in, first out - FIFO***).

В первом случае необходимо, чтобы система виртуальной памяти отслеживала, когда страница использовалась последний раз. Если нужно загрузить новую страницу, то на диск выгружается та страница, которая не использовалась дольше всех остальных, а ее страничный фрейм освобождается. В соответствии со вторым алгоритмом, на диск перемещается та страница, которая дольше всех находилась в памяти, независимо от того, как часто она использовалась.

Кроме того, стратегии замещения делятся на ***локальные*** (***local replacement policy***) и ***глобальные*** (***global replacement policy***). При стратегии локального замещения каждому процессу выделяется фиксированное (динамически настраиваемое) число страничных фреймов. Когда процесс использовал все выделенные ему фреймы, система виртуальной памяти удаляет из физической памяти одну из его страниц при каждой страничной ошибке в данном процессе. В случае стратегии глобального замещения для обработки страничной ошибки используется любой страничный фрейм, независимо от того, принадлежит ли он процессу, в котором произошла эта страничная ошибка. Так при использовании стратегии глобального замещения FIFO необходимо отыскать страницу, которая дольше остальных находилась в памяти, а при использовании стратегии локального замещения FIFO необходимо отыскать самую старую страницу данного процесса.

Стратегия глобального замещения создает ряд проблем:

- процессы подвержены влиянию других процессов;

- плохо·е приложение может подорвать работу всей системы.

По этим причинам диспетчер виртуальной памяти Windows использует стратегию локального замещения FIFO. Для реализации этого подхода ему необходимо для каждого процесса отслеживать его страницы, находящиеся в памяти. Это множество страниц называется ***рабочим набором*** (***working set***) процесса.

В момент создания процессу назначается минимальный размер рабочего набора, т.е. минимальное число страниц процесса, которые будут гарантированно находиться в памяти во время его исполнения. Если память не слишком загружена, то диспетчер виртуальной памяти позволяет процессу иметь в памяти число страниц, равное максимальному размеру его рабочего набора. Если процессу требуются дополнительные страницы, то диспетчер виртуальной памяти удаляет одну из его страниц при каждой сгенерированной процессом страничной ошибке.

Замещаемые страницы рабочего набора на самом деле еще некоторое время остаются в памяти после их замещения, их можно быстро вернуть в рабочий набор, и это не требует считывания с диска.

Когда физической памяти становится недостаточно, диспетчер виртуальной памяти использует ***автоматическое урезание рабочего набора*** (***automatic working set trimming***), для увеличения объема свободной памяти. Он просматривает все процессы и сравнивает текущий размер рабочего набора с минимальным. Когда он обнаруживает процессы, рабочие наборы которых больше минимального, он удаляет страницы из их рабочих наборов.

После того, как рабочий набор процесса уменьшился до минимума, диспетчер виртуальной памяти отслеживает число генерируемых данным процессом страничных ошибок. Если их много, а память не слишком загружена, то диспетчер виртуальной памяти увеличивает размер рабочего набора. Однако если в течение некоторого времени, в процессе не происходит страничных ошибок, то либо код, исполняемый потоками процесса, комфортно умещается в минимальном рабочем наборе процесса, либо ни один из потоков процесса не исполняется. Например, процесс регистрации пользователя в системе просто ждет, пока позьзователь зарегистрируется, а после регистрации пользователя, этот процесс ждет выхода пользователя из системы. Для процессов, которые простаивают большую часть времени, диспетчер виртуальной памяти продолжает уменьшать размер рабочего набора, пока в процессе не произойдет страничная

ошибка. Это будет означать, что потоки процесса вновь начали исполняться, либо достигнут минимаьный размер памяти, необходимый потокам процесса для исполнения.

Процесс может изменить минимальный и максимальный размер своего рабочего набора, вызвав сервис объекта процесс. Однако диспетчер виртуальной памяти, используя стратегии локального замещения и автоматического урезания рабочего набора, пытается обеспечить максимально возможную производительность для каждого процесса.

База данных страничных фреймов

Таблицы страниц процесса содержат информацию о том, в каком месте физической памяти расположены виртуальные страницы. Кроме того, диспетчеру виртуальной памяти нужна структура данных для отслеживания состояния физической памяти. Чтобы знать, свободен ли данный страничный фрейм или, если нет, то кто его использует.

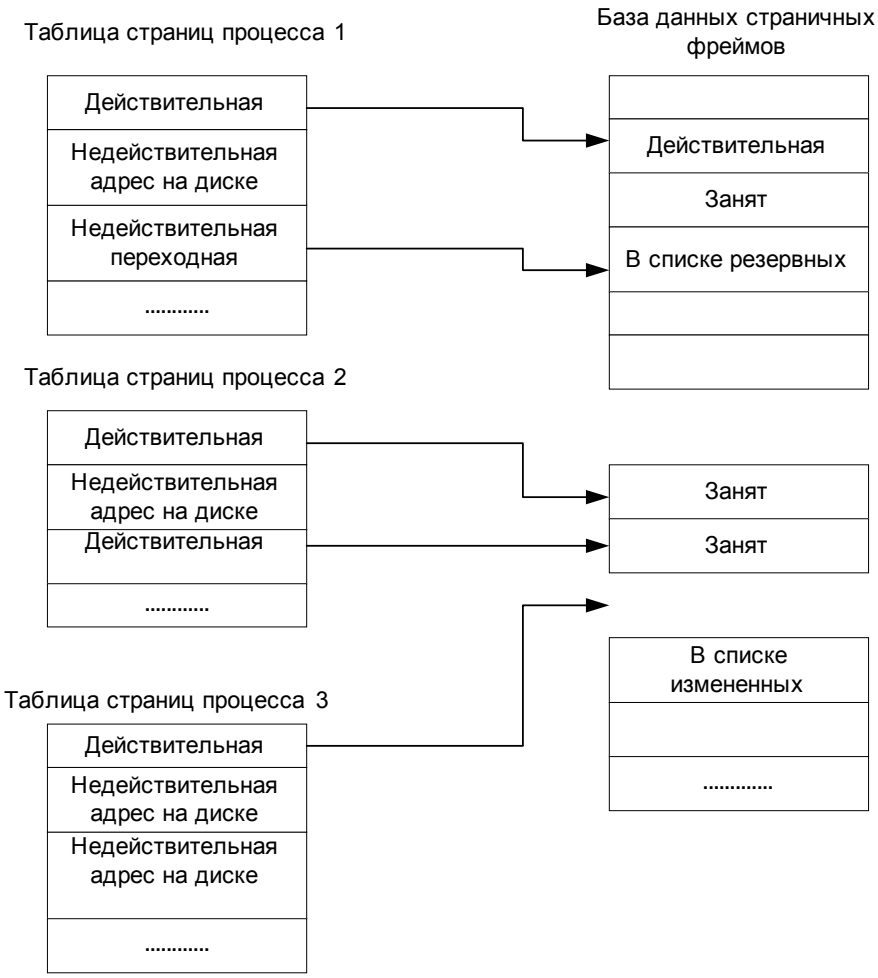
Для этого используется ***база данных страничных фреймов*** (***page frame database***). Она представляет собой массив из элементов от нуля до числа страничных фреймов в системе минус 1. Каждый элемент содержит информацию о соответствующем страничном фрейме. База данных страничных фреймов и ее связь с таблицами страниц показана на рисунке (*Рисунок 6.6*).

Действительные элементы таблицы страниц указывают на элементы базы данных страничных фреймов. Диспетчер виртуальной памяти использует этот указатель, когда процесс обращается по действительному виртуальному адресу, чтобы найти физическую страницу, соответствующую виртуальному адресу.

Некоторые недействительные элементы таблицы страниц также ссылаются на элементы базы данных страничных фреймов. Эти переходные элементы таблицы страниц указывают на страничные фреймы, которые могут быть, но еще не использованы повторно и их содержимое пока не изменилось. Если процесс обращается к одной из таких страниц, прежде чем ее использовал другой процесс, диспетчер виртуальной памяти может ее быстро восстановить.

Другие недействительные элементы таблицы страниц содержат адреса на диске, по которым хранятся страницы. При обращении процесса к ним происходит страничная ошибка, и диспетчер виртуальной памяти считывает содержимое страницы с диска.

Рисунок 6.6 Таблицы страниц и база данных страничных фреймов



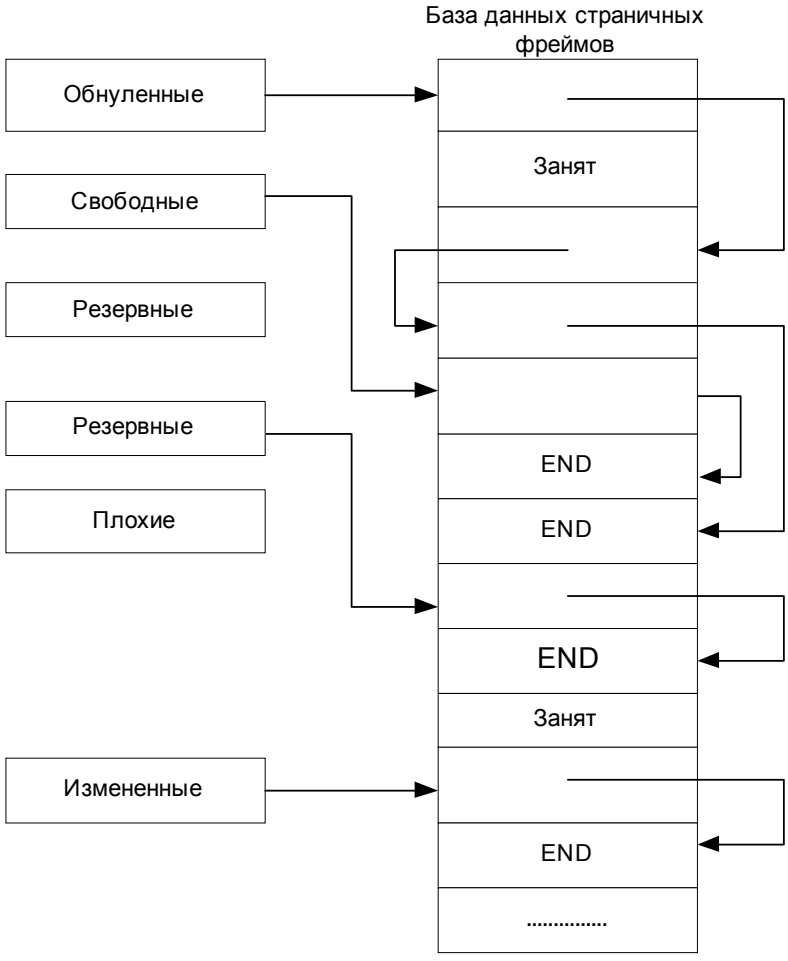
В любой момент времени страничный фрейм может находиться в одном из шести состояний:

- ***Действительный*** - страничный фрейм используется процессом и на него указывает действительный элемент таблицы страниц;
- ***Обнуленый*** - страничный фрейм свободен и инициализирован нулями;
- ***Свободный*** - страничный фрейм свободен, но не инициализирован
- ***Резервный*** - данный фрейм использовался процессом, но был удален из его рабочего набора. Соответствующий элемент таблицы страниц недействителен, но помечен как переходный;

- ***Измененный*** - аналогично резервному, но процесс, использовавший данный страничный фрейм, осуществил запись в него и содержимое еще не записано на диск. Соответствующий элемент таблицы страниц недействителен, но помечен как переходный;
- ***Плохой*** - страничный фрейм вызвал ошибку четности или другой аппаратный сбой, и его нельзя использовать.

В базе данных страничные фреймы, находящиеся в одном и том же состоянии, группируются и получается пять отдельных списков: список обнуленных, список свободных, список резервных, список измененных и список плохих страниц. Связь между базой данных страничных фреймов и списками страниц показана на рисунке (*Рисунок 6.7*).

Рисунок 6.7 Списки страниц в базе данных страничных фреймов

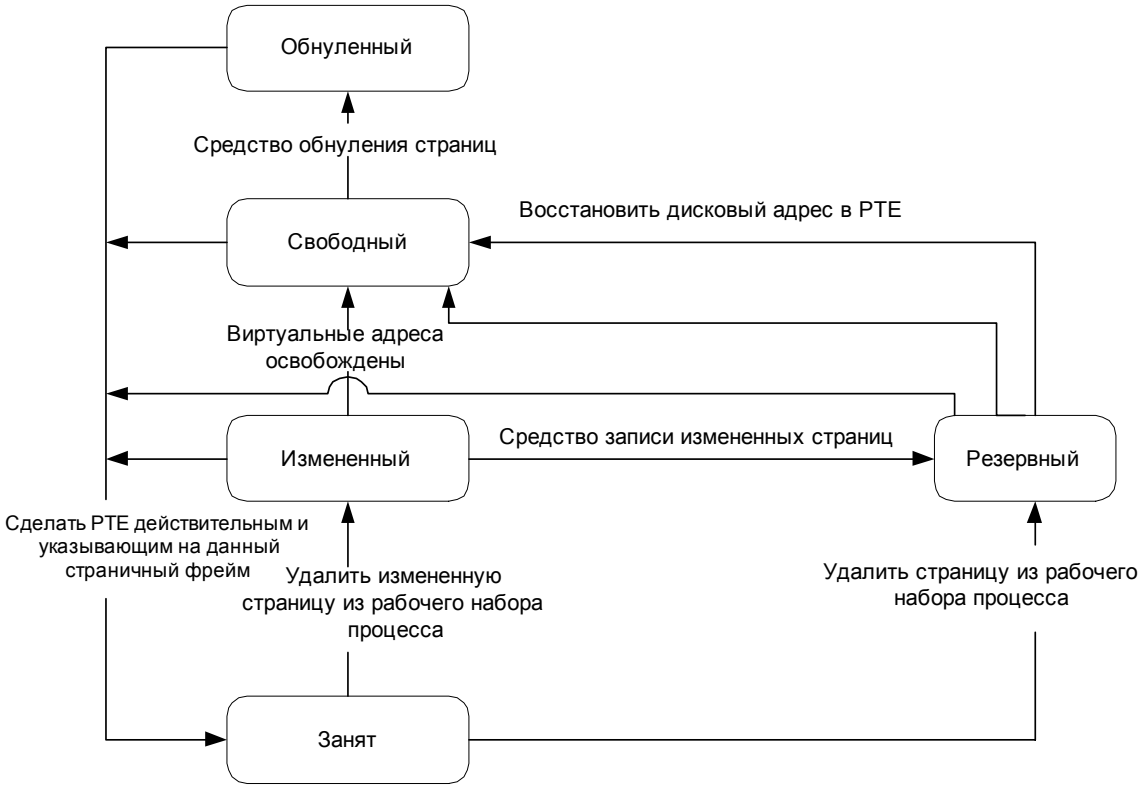


В данные списки включены все страничные фреймы, которые не используются в данный момент. Указатели, на используемые фреймы, содержатся в таблице страниц соответствующего процесса. Если процесс освобождает страничный фрейм, то диспетчер виртуальной памяти помещает его обратно в один из списков.

Если диспетчеру виртуальной памяти требуется инициализированный нулями страничный фрейм для обработки страничной ошибки, то он пытается взять первый из списка обнуленных, если этот список пуст, то диспетчер выбирает фрейм из списка свободных и обнуляет его. Если диспетчеру не нужна инициализированная страница, то он берет первую из списка свободных; если этот список пуст, то используется первая из списка обнуленных.

Если оба списка пусты, то используется список резервных. Когда количество страниц в списках обнуленных, свободных и резервных снижается до порогового значения, поток под названием ***средство записи измененных страниц*** (***modified page writer***) пробуждается и записывает содержимое измененных страниц на диск, после чего помещает их в список резервных (см. *Рисунок 6.8*).

Рисунок 6.8 Диаграмма состояний страничных фреймов



Дескрипторы виртуальных адресов

Момент загрузки страниц в память диспетчер памяти определяет, используя алгоритм подкачки по требованию. Страница загружается с диска, если поток, обращаясь к ней, вызвал ошибку страницы. Подкачка по требованию является одной из форм ***отложенных вычислений*** (***lazy evalution***), операция выполняется только при ее абсолютной необходимости.

Диспетчер памяти использует этот же механизм и при формировании таблиц страниц. Например, когда поток передает с помощью ***VirtualAlloc***, диспетчер памяти мог бы немедленно создать таблицы страниц, необходимые для доступа ко всему объему выделенной памяти. Вместо этого диспетчер виртуальной памяти откладывает формирование таблицы страниц до тех пор, пока поток не вызовет ошибку страницы. Как же тогда диспетчер виртуальной памяти определяет, какие адреса свободны? Чтобы решить эту проблему диспетчер поддерживает набор структур данных, которые позволяют вести учет зарезервированных и свободных виртуальных адресов в адресном пространстве процесса. Эти структуры данных называются ***дискрипторами виртуальных адресов*** (***virtual address descriptors, VAD***). Для каждого процесса диспетчер памяти поддерживает свой набор VAD, описывающий состояние адресного пространства этого процесса.

При первом обращении потока по какому-либо адресу диспетчер памяти должен создать PTE страницы, содержащий данный адрес. Для этого он находит VAD, чей диопазон включает нужный адрес, и использует его информацию для заполнения PTE. Если адрес выпадает из диопазонов VAD или находится в зарезервированном, но не переданном диапазоне адресов, диспетчер памяти узнает, что поток не выделил память до попытки ее использования, и генерирует нарушение доступа.

ЛАБОРАТОРНАЯ РАБОТА 6

ВИРТУАЛЬНАЯ ПАМЯТЬ

Теория

Виртуальная память

Страничная виртуальная память используется в большинстве современных операционных систем. В этом случае каждый процесс имеет свое собственное виртуальное адресное пространство, которое он использует для ссылки на различные объекты, обычно находящиеся в памяти. Часть виртуального адресного пространства определяется редактором связей, когда он создает выполняемый модуль (*exe*), остальная часть адресного пространства может быть определена динамически во время выполнения, используя технику, описываемую в этой лабораторной работе. После того, как статическая часть виртуального адресного пространства сконструирована, она хранится во вторичной памяти (страничном файле).

В современном компьютере процесс может выполнять инструкции или загружать данные, которые расположены в первичной памяти (RAM). Первичная память маленькая и быстрая по сравнению со вторичной памятью. Она также значительно дороже. В большинстве компьютеров не хватит первичной памяти, даже для хранения виртуального адресного пространства одного процесса. Но в то же время во вторичной памяти имеется достаточно места для хранения виртуальных адресных пространств нескольких процессов. Однако следует учитывать разницу в скорости - запись/чтение байта в первичной памяти требует пару циклов процессора, а во вторичной - тысячи циклов процессора.

Для экономии первичной памяти система виртуальной памяти загружает только части виртуальных адресных пространств процессов. Поток выполняется в адресном пространстве процесса, при его выполнении только та часть виртуального адресного пространства, которая используется в данный момент, загружается в первичную память, тогда как другие части остаются во вторичной памяти. Когда процесс больше не нуждается в части виртуального адресного пространства (на какое-то время), эта часть (порция) копируется обратно во вторичную память. Это позволяет участки первичной

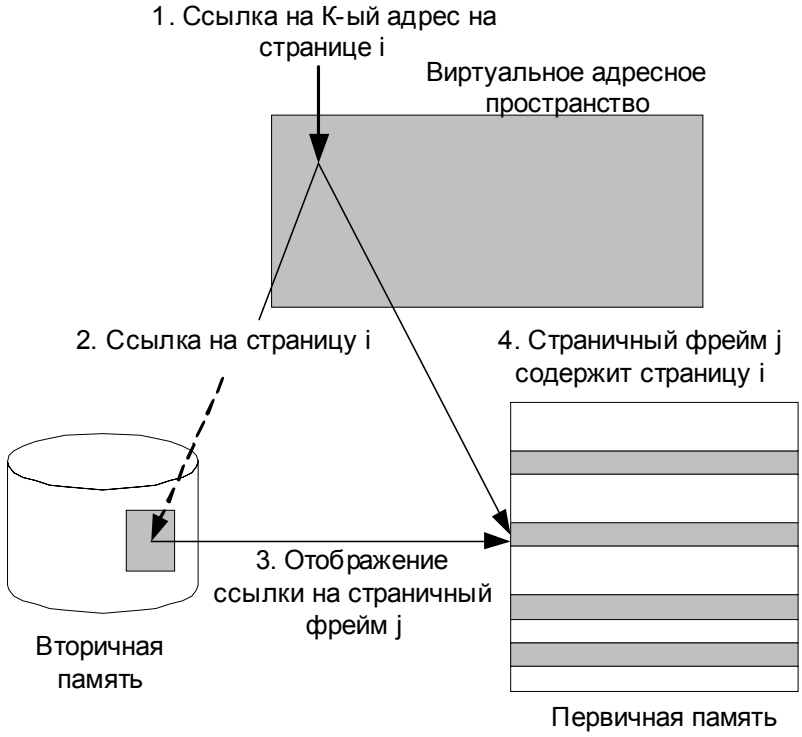
памяти, используемые для хранения одних частей виртуального адресного пространства, в следующий момент времени использовать для хранения других частей.

Для эффективного перемещения данных из первичной памяти во вторичную и обратно используется копирование блоков.

Вторичная память - блочная. Если необходимо слово, то для его получения необходимо прочитать целый блок (т.е. в первичную память перемещается целый блок). Загружать целые блоки, а не отдельные слова выгоднее. Когда поток обращается (ссылается) на i существует большая вероятность, что он дальше будет ссылаться на i+1. Страничная виртуальная память загружает и выгружает блоки фиксированного размера, называемые ***страницами***, между первичной и вторичной памятью. Границы страниц прозрачны для программиста.

На рисунке (*Рисунок 6.9*) приводится общая схема организации страничной виртуальной памяти.

Рисунок 6.9 Страничная система



Когда поток обращается по виртуальному адресу **k** (шаг 1) , система виртуальной памяти сначала определяет номер страницы, содержащий виртуальный адрес **k** (шаг 2). Если эта страница уже загружена в

первичную память (шаг 3), то система виртуальной памяти транслирует виртуальный адрес в физический (тот страничный фрейм, где размещена эта страница).

Если страница не загружена в первичную память, то нормальное выполнение потока прерывается до тех пор, пока страница не будет загружена в страничный фрейм, а после этого возобновляется (шаг 4). Ссылка на виртуальный адрес **k** приводит к обращению к физическому адресу первичной памяти, куда загружена соответствующая страница.

Преимущества использования виртуальной памяти в том, что несколько процессов могут использовать первичную память одновременно, несмотря на то, что их виртуальные адресные пространства значительно больше, чем объем первичной памяти компьютера. Но за это приходится расплачиваться тем, что выполняющийся поток будет ждать пока нужная в данный момент часть его виртуального адресного пространства, будет загружено в первичную память.

Windows NT использует страничную систему виртуальной памяти. Существует несколько уникальных особенностей виртуальной памяти NT, поэтому далее мы рассмотрим именно реализацию NT.

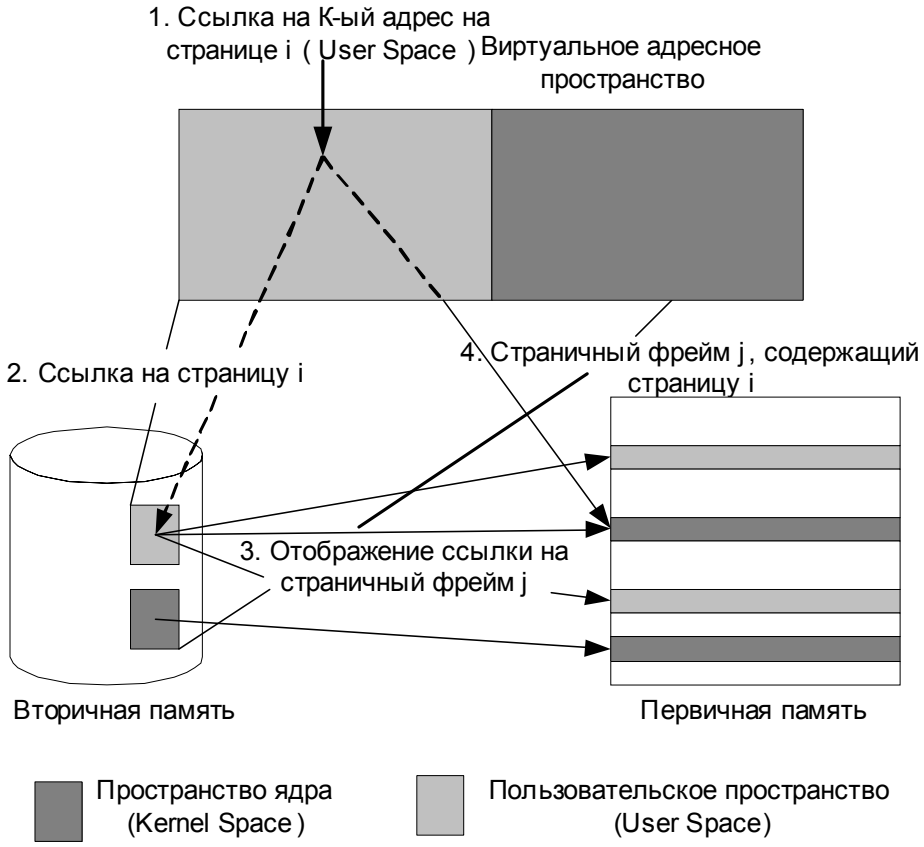
Модель виртуальной памяти Windows

Каждый процесс в Windows получает фиксированное (по размеру) виртуальное адресное пространство - 4 Гб, которое конечно много больше, чем объем первичной памяти (RAM) в современных компьютерах. Процессу не требуется использовать все виртуальное адресное пространство, только сколько нужно. Обычно exe файл - программ намного меньше, чем адресное пространство. Как показано на (*Рисунок 6.10*) часть виртуального адресного пространства, обычно 2 Гб, используется пользователем, а оставшаяся часть 2 Гб - системой.

Хотя системная область адресного пространства существует в виртуальном адресном пространстве процесса, к ней может обращаться только поток, выполняющийся в режиме ядра.

В нормальной конфигурации первые 64 K (адреса с 0X00000000 до 0X0000FFFF) виртуального адресного пространства не используются. Это сделано для того, чтобы, если программист пишет код с ошибочными указателями, они обычно 0 или маленькие. Система виртуальной памяти, если программа пытается обратиться по адресу первых 64 K, выдает ошибку. Первая используемая часть адресного пространства начинается с адреса 0X00010000.

Рисунок 6.10 Страничная система Windows



Операционная система нуждается в средствах определения, какое количество адресного пространства процесс будет действительно использовать. Редактор связей строит статический образ - exe файл, который и будет определять часть адресного пространства, которое будет использовано для кода. DLL и другие динамически размещаемые порции адресного пространства будут добавлены к виртуальному адресному пространству во время выполнения.

Два этапа динамического добавления адресов к адресному пространству процесса:

- ***резервирование*** порции адресного пространства (региона);
- ***передача*** блока страниц (региона) в адресное пространство.

На первом этапе поток динамически резервирует регион виртуальных адресов без действительной записи во вторичную память - страничный файл (файл страниц). Поток процесса может также освободить регион адресов, который предварительно зарезервировал.

На втором этапе предварительно зарезервированные адреса передаются. Передаваемая порция размещается в страничном файле. Когда поток процесса обращается (ссылается) по переданному адресу, то страница, содержащая этот адрес, будет загружена из страничного файла в первичную память. При первом обращении будет загружена инициализированная 0 страница.

Каждый процессор поддерживает ***гранулярность выделения памяти*** (***allocation granularity***) - min размер блока адресов, который может быть зарезервирован. Обычно 64 K (однако вы можете использовать ***GetSystemInfo*** для определения этого размера). Перед выполнением резервирования адрес автоматически изменяется с учетом этого размера.

Каждый процессор также поддерживает собственный размер страницы. Обычно 4K или 8K (`GetSystemInfo`). Память передается наборами страниц, таким образом, действительно передаваемые адреса могут быть много меньше, чем зарезервированные. После того как адрес передан, поток может использовать эту память.

Страничная система изнутри

Процесс трансляции виртуальных адресов в физические адреса представлен на рисунке (*Рисунок 6.11*). При трансляции адресов используются аппаратные решения для выявления откачанных страниц и быстрого отображения их в страничные фреймы. NT использует 2-х уровневую трансляцию адресов:

- ***Byte index*** - смещение на странице;
- ***PTE page table index*** - смещение в таблице;
- ***PDE page directory index*** - смещение в каталоге.

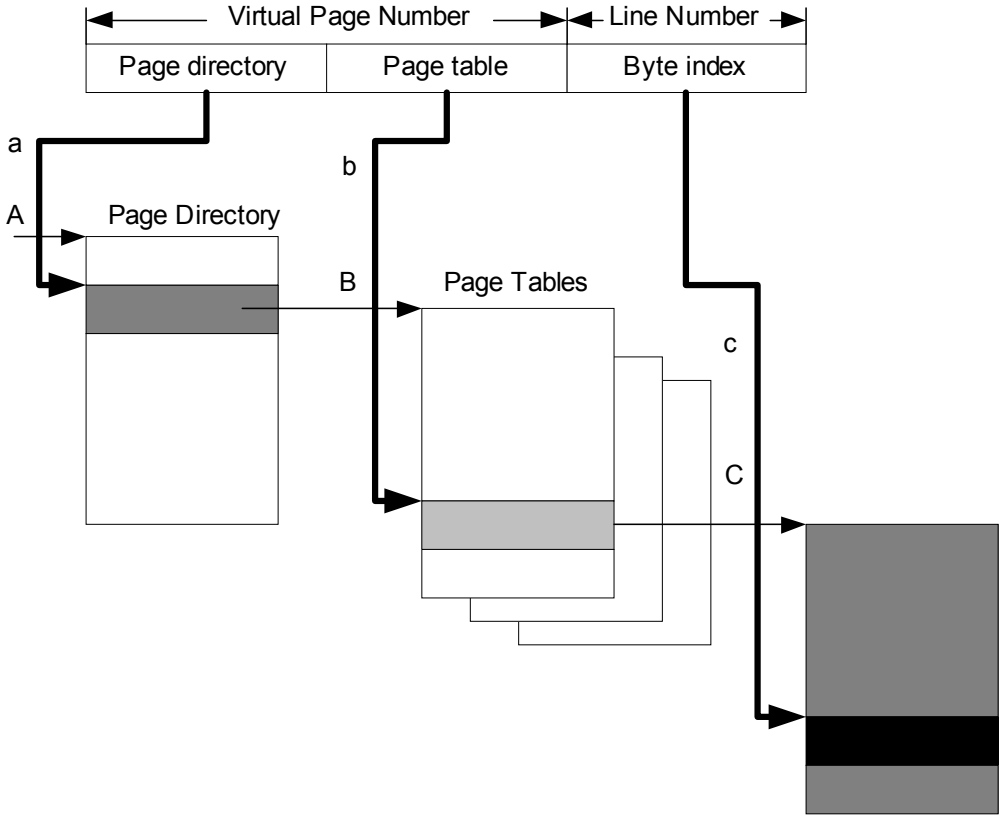
1. Описатель процесса содержит указатель A на начало каталога страниц данного процесса (page directory)
2. (PDE) page directory index - это смещение в каталоге страниц, где размещен указатель на начало таблицы страниц
3. Каждый процесс имеет несколько различных таблиц страниц. PDE ссылается на начало определенной таблицы страниц
4. PTE - это смещение в таблице страниц относительно адреса определенной таблицы
5. Искомая таблица находится в первичной памяти в страничном фрейме j, PTE указывает на страничный фрейм. Если страница не загружена ***диспетчер виртуальной памяти*** (***Virtual Memory***

Manager) должен разместить страницу в страничном файле, найти страничный фрейм, выделить его процессу, а затем загрузить страницу в этот фрейм.

6. Окончательно byte index прибавляется к базовому адресу страничного фрейма, чтобы получить нужный адрес в первичной памяти.

Хотя page directory может быть расположена в любом месте (указатель A - см. *Рисунок 6.11*), на практике она размещена в фиксированном месте адресного пространства 0XC0300000 (x86) и 0XC018000 (Alpha). Оба этих процессора используют соответствующий регистр для ссылки на page directory. Этот регистр сохраняется как часть контекста потока, когда новый поток назначается на процессор.

Рисунок 6.11 Трансляция адресов



NT использует различные таблицы страниц, чтобы различать используемое адресное пространство. Главное различие между страницами - одни пользовательские, а другие ядра, они отображаются в различных таблицах страниц. Т.к. страницы ядра хранятся в отдельной таблице, различные процессы имеют PDEs для области ядра,

указывающие на те же таблицы. Другое различие - некоторые части области ядра имеют страничную организацию как и пользовательские, а другие не страничную организацию.

Каждое PDE ссылается на номер страничного фрейма, когда соответствующая страница загружена. Существует также набор флагов, описывающих соответствующие страницы (доступен ли PTE, зарезервирована ли страница, записаны ли на нее новые значения с тех пор как, она последний раз загружена в первичную память). Требуется, чтобы в page directory был PDE, а в page table был PTE. Современные компьютеры используют ассоциативную память, называемую translation look-aside buffer (TLB). Коротко TLB это кэш-память для недавно транслированных PDEs и PTEs. Поиск во всей TLB проводится за 1 цикл.

NT использует ***подкачку страниц по запросу*** (***demand paging***) - это означает, что страницы не будут загружены в первичную память до тех пор, пока на них не сошлются (к ним не обратятся). PTEs даже не создаются до тех пор, пока соответствующая страница не загружена. Ведь процесс может зарезервировать память, но никогда ее не использовать, он может даже передать страницы памяти, но никогда не обратиться к ним во время выполнения. Многие PTEs могли бы быть созданы и никогда бы не использовались, следовательно, пустая трата памяти для PTEs. Так как PTE не создается до тех пор, пока они не используются в первый раз, OS должна подкачать другую страницу данных, чтобы хранить информацию о том, что зарезервировано и что передано. Это - ***virtual address descriptor*** (***VAD***). Он создается, когда процесс резервирует или передает виртуальные адреса. Когда поток первый раз ссылается на адрес в пределах VAD, PTE создается, чтобы выполнить трансляцию адреса нормально.

NT использует ***рабочие наборы*** (***working sets***). Рабочий набор процесса:

- минимальный 20 - 50 страниц;
- максимальный 45 - 345 страниц (максимум может быть изменен системным администратором).

Функции виртуальной памяти

Win32 API предоставляет несколько функций для работы с виртуальной памятью. **`GetSystemInfo`**, **`GlobalMemoryStatus`** и **`VirtualQuery`** используются для определения состояния виртуальной памяти процесса.

```
VOID GetSystemInfo(
LPSYSTEM_INFO lpSystemInfo
);

typedef struct_SYSTEM_INFO {
DWORD dwOemId;
DWORD dwPageSize;
LPVOID lpMinimumApplicationAddress;
LPVOID lpMaximumApplicationAddress;
DWORD dwActiveProcessorMask;
DWORD dwNumberOfProcessors;
DWORD dwProcessorType;
DWORD dwAllocationGranularity;
DWORD dwReserved;
} SYSTEM_INFO,*LPSYSTEM_INFO;
```

Эта функция заполняет **lpSystemInfo** , чтобы проинформировать вас о различных установках виртуальной памяти. **dwPageSize** - размер страницы, **lpMinimumApplicationAddress** - нижний адрес, который можно использовать без ошибок памяти. **lpMaximumApplication Address** - старший адрес, который может использовать приложение. Функция **VOID GlobalMemoryStatus (LPMEMORYSTATUS lpBuffer)** возвращает более детальную информацию о виртуальной памяти в структуре **MEMORYSTATUS**.

```
typedef struct_MEMORYSTATUS {
DWORD dwLength;
DWORD dwMemoryLoad;
DWORD dwTotalPhys;
DWORD dwAvailPhys;
DWORD dwTotalPageFile;
DWORD dwAvailPageFile;
DWORD dwTotalVirtual;
DWORD dwAvailVirtual;
} MEMORYSTATUS, *LPMEMORYSTATUS;
```

dwMemoryLoad параметр - это значение между 0 и 100, которая отражает относительную утилизацию физической памяти. Это число полезно только, если вы часто его используете и интуитивно чувствуете.

dwTotalPhys - общий объем физической памяти, сконфигурированный в машине.

dwAvailPhys - тот объем физической памяти, который не занят рабочими наборами - доступен для использования.

dw Total Page File - общее число байт, которые можно сохранять в файле страниц, а **dwAvail PageFile** - число байт, которые доступны в файле страниц (свободны).

dwTotalVirtual - размер пользовательской части всего адресного пространства.

dwAvailVirtual - число доступных адресов.

Последняя функция используется для определения состояния виртуальной памяти **VirtualQuery**.

```
DWORD VirtualQuery(
LPCVOID lpAddress,
PMEMORY_BASIC_INFORMATION lpBuffer,
DWORD dwLength
);
```

lpAddress - это адрес арбитр в адресном пространстве, перед использованием он устанавливается с учетом нижней границы страницы. Соответствующие страницы принадлежат к набору страниц в блоке, и все будут иметь одинаковые свойства, заданные при передаче. Вызов **VirtualQuery** с любым адресом в блоке будет возвращать будет возвращать одинаковую информацию в структуре **MEMORY_BASIC_INFORMATION** для каждого адреса в блоке.

```
typedef struct_MEMORY_BASIC_INFORMATION {
PVOID BaseAddress;
PVOID AllocationBase;
DWORD AllocationProtect;
DWORD RegionSize;
DWORD State;
DWORD Protect;
DWORD Type;
}MEMORY_BASIC_INFORMATION, *PMEMORY_BASIC_INFORMATION;
```

BaseAddress - адрес первой страницы в блоке, а **AllocationBase** - адрес начала региона. Поле **AllocationProtect** задает разрешенный доступ к страницам в блоке (только для чтения, для чтения и записи, для выполнения и т.д.). **RegionSize** - число байтов в блоке (блок страниц, входящих в регион). Поле состояния может принимать значения **MEM_COMMIT**, **MEM_RESERVE** или **MEM_FREE**. **Protect** определяет тип защиты страниц в блоке. **Type** определяет тип страниц, которые будут сохраняться в регионе (**MEM_IMAGE**, **MEM_MAPPED**, **MEM_PRIVATE**).

Расширенная версия **VirtualQuery** позволяет потоку запросить блок виртуальной памяти у другого процесса.

```
DWORD VirtualQueryEx(
HANDLE hProcess,
LPCVOID lpAddress,
PMEMORY_BASIC_INFORMATION lpBuffer,
DWORD dwLength
);
```

Описатель (***Handle***) вызывающего потока должен иметь доступ **PROCESS_QUERY_INFORMATION** к удаленному процессу.

Функции **VirtualAlloc** и **VirtualFree** используются для динамического резервирования, передачи и освобождения памяти.

```
LPVOID VirtualAlloc(
LPVOID lpAddress,
DWORD dwSize,
DWORD flAllocationType,
DWORD dwProtect
);
```

VirtualAlloc используется и для резервирования и для передачи (необходимо помнить, что граница региона определяется единицами выделения памяти системой, а граница блока определяется границами страницы).

flAllocationType определяет различные типы операции выделения памяти - **MEM_COMMIT** (блок передан); **MEM_RESERVE** (операция резервирования региона); вы можете комбинировать **MEM_COMMIT** и **MEM_RESERVE** при вызове, чтобы регион был зарезервирован и каждая страница региона была бы передана.

MEM_RESET определяет, что блок страниц не будет сохранен в файле страниц при перезаписи. **flProtect** определяет защиту по доступу. **VirtualAllocEx** позволяет одному процессу резервировать и передавать адреса в адресное пространство другого процесса.

VirtualFree и **VirtualFreeEx** используются для освобождения адресного пространства.

```
LPVOID VirtualFree(
LPVOID lpAddress,
DWORD dwSize,
DWORD dwFreeType
);
```

Существует 5 функций, используемых для ***блокировки*** (***"lock"***) страниц в первичной памяти. Такие страницы не могут быть переписаны системой виртуальной памяти во вторичную память.

```
BOOL VirtualLock(
LPVOID lpAddress,
DWORD dwSize
);
```

Мы блокируем переданный до этого блок страниц в первичной памяти. Адреса страниц - в пределах от **lpAddress** до **lpAddress + dwSize**. Максимум 30 страниц могут быть заблокирован в первичной памяти.

Если физической памяти не хватает, то заблокированные страницы только ухудшают эту ситуацию, т.к. они не дают возможность работать механизму рабочих наборов. В будущем страницы будут

заблокированы, даже когда процесс-владелец бездействует (но блокировка будет снята, когда процесс закончится). Если у процесса привилегия **SE_INC_BASE_PRIORITY_NAME**, то это позволяет увеличивать его рабочий набор, используется **SetProcessWorkingSetSize.**

Заблокированные страницы можно разблокировать используя

```
BOOL VirtualUnlock(
LPVOID lpAddress,
DWORD dwSize
);
```

Много потоков могут вызвать **VirtualLock**, но только один вызвавший **VirtualUnlock** разблокирует блок страниц.

Если вы хотите, чтобы каждой операции блокирования соответствовала операция разблокирования, вы должны использовать **GlobalLock** и **GlobalUnlock** или **LocalLock** и **LocalUnlock** для управления объектами локальной и глобальной памяти соответственно.

ReadProcessMemory и **WriteProcessMemory** могут использовать для чтения/записи одним процессом памяти другого процесса, в случае, если вы получили описатель этого процесса с соответствующими привилегиями. Это мощный механизм совместного использования памяти (см. **MSDN Home Page -http://msdn.microsoft.com/**) .

Чему нужно научиться?

Нужно научиться эффективно, использовать возможности, предоставляемые системой виртуальной памяти.

Задания

Уровень 1 (A)

Используйте функцию **VirtualAlloc** для того, чтобы сначала **зарезервировать** регион, состоящий из некоторого количества страниц, а затем осуществлять **передачу** по одной странице. Используйте обработку исключений при доступе к странице. Если возникает **ошибка страницы**, то нужно передать следующую страницу из зарезервированного региона.

Скелет кода 1 (A)

```
#define PAGELIMIT 10 // Количество страниц

LPTSTR lpNxtPage; // адрес следующей запрашиваемой страницы
DWORD dwPages = 0; // счетчик страниц
```

```
DWORD dwPageSize; // размер страницы

INT PageFaultExceptionFilter(DWORD dwCode)
{
  LPVOID lpvResult;

// Проверить, что ошибка страницы (иначе выход)

….
Printf("Ошибка страницы\n");


  // если лимит исчерпан, то выход

  if (dwPages >= PAGELIMIT)
    {
      printf("Лимит исчерпан\n");
      return 0;
    }

  // иначе передать следующую страницу

  lpvResult = VirtualAlloc(
                           (LPVOID) lpNxtPage,
                           dwPageSize,
                           MEM_COMMIT,
                           PAGE_READWRITE);
  if (lpvResult == NULL )
    {
      printf("Ошибка при вызове VirtualAlloc\n");
      return 0;
    } else {
      printf ("Передана следующая страница\n");
    }


  // Увеличить счетчик страниц и передвинуть указатель на следующую
страницу.

…
  return 0;
}
VOID main(VOID)
{
  LPVOID lpvBase; // базовый адрес выделенной памяти
  LPTSTR lpPtr; // указатель на текущую страницу


  SYSTEM_INFO sSysInfo;


  GetSystemInfo(&sSysInfo);

  printf ("Размер страницы %d.\n", sSysInfo.dwPageSize);
  printf("Процессор %d\n",sSysInfo.dwProcessorType);
  printf("Число процессоров %d\n",sSysInfo.dwNumberOfProcessors);
```

```
  dwPageSize = sSysInfo.dwPageSize;

  // Зарезервировать страницы в виртуальном адресном пространстве процесса

  lpvBase = VirtualAlloc(
                        NULL,
                        PAGELIMIT*dwPageSize,
                        MEM_RESERVE,
                        PAGE_NOACCESS);
  if (lpvBase == NULL )
  { printf("Ошибка при резервировании\n"); exit 0; }

  lpPtr = lpNxtPage = (LPTSTR) lpvBase;

  // Использовать обработку исключений при доступе к странице
  // Если возникает ошибка страницы
  // то передать следующую страницу из зарезервированного региона

  for (i=0; i < PAGELIMIT*dwPageSize; i++)
    {
      __try
        {
          // Запись в память

          lpPtr[i] = 'A';
           printf("Записали в память A!\n");
        }

      // Если ошибка страницы, то передать страницу и попробовать снова

      __except ( PageFaultExceptionFilter( GetExceptionCode() ) )
        {
          …
          ExitProcess( GetLastError() );
}

    }

  // Освободить выделенную память


  bSuccess = VirtualFree(
                        lpvBase,
                        dwPages*dwPageSize,
                        MEM_DECOMMIT| MEM_RELEASE);
// Проверить успешно ли освободили память
}
```

Уровень 2, 3 (А)

Вы должны создать процесс с двумя потоками:

- первым потоком для симуляции работы с виртуальной памятью;
- и вторым потоком для мониторинга того, что будет происходить с памятью в результате работы первого потока.

Ваш поток - симулятор будет читать файл, который будет содержать по одной записи для каждой операции с виртуальной памятью в следующем формате:

- ***Время*** (время в миллисекундах от начала старта процесса) - в это время необходимо выполнить заданную операцию;
- ***Регион/Блок*** (номер региона при резервировании или номер блока при передаче, т.е. в зависимости от операции);
- ***Операция*** . Это поле может принимать следующие значения:
  - 1 - зарезервировать регион,
  - 2 - передать блок,
  - 3 - не сохранять блок в страничном файле при его изменении,
  - 4 - освободить регион,
  - 5 - вернуть блок,
  - 6 - блокировать блок,
  - 7 - снять блокировку;
- ***Размер*** (размер региона/блока в байтах);
- ***Доступ.*** Это поле может принимать следующие значения:
  - **PAGE_READONLY**,
  - **PAGE_READWRITE**,
  - **PAGE_EXECUTE**,
  - **PAGE_EXECUTE_READ**,
  - **PAGE_EXECUTE_READWRITE**.

Когда симулятор читает запись, он вызывает соответствующую функцию виртуальной памяти, используя заданные параметры. А ваш поток - монитор должен спать определенное время, просыпаться и проверять состояние виртуальной памяти. Он должен записывать состояние памяти каждый раз при проверке. Сформировать отчет со следующей информацией:

- размер страницы
- гранулярность
- состояние физической памяти (Physical Memory)
- состояние виртуальной памяти (Virtual Memory)

- состояние страничного файла (Page file).

Посмотреть, как информация об использовании памяти отображается в Task Manager и в Process Viewer (pview.exe) и сделать так же.

Скелет кода 2, 3 (A)

Процесс должен быть с двумя потоками. Первый поток читает файл сценарий и выполняет заданные в нем действия с виртуальной памятью, а второй отслеживает поведение первого.

```
int main(int argc, char* argv[])
{
// Инициализация
theClock=0;

// Открыть файл сценария и файл отчета
scriptFID=fopen(argv[1],"r");
reportFID=fopen(REPORT_FILE,"w");

// Создать поток
while( fscanf(…) != EOF )
{
if( time >= theClock ) {
if( time > theClock )
simActivity( time - theClock, NULL, 0 );
theClock=time; }
//Считать команду из файла сценария и выполнить ее

switch( vmOp )
{

case 1: // Reverse a region
break;
case 2: // Commit a block
break;
case 3: // Reset a block
break;
case 4: // Release a region
break;
case 5: // Decommit a block
break;
case 6: // Lock a block
break;
case 7: // Unlock a block
break;


} // switch
}//while
}

void simActivity(int period, LPVOID lpAddress, DWORD dwSize)
{
#ifdef REALTIME
```

```
Sleep(period);
#else
Sleep(100);
#endif
if(lpAdddress != NULL )
ZeroMemory(lpAdress, dwSize);
}
```

Поток Монитор

```
DWORD WINAPI monitorThread(LPVOID fid)
{
//
while(…)
{
// Сбор информации
}
}
```

Глава

7

Совместное использование памяти

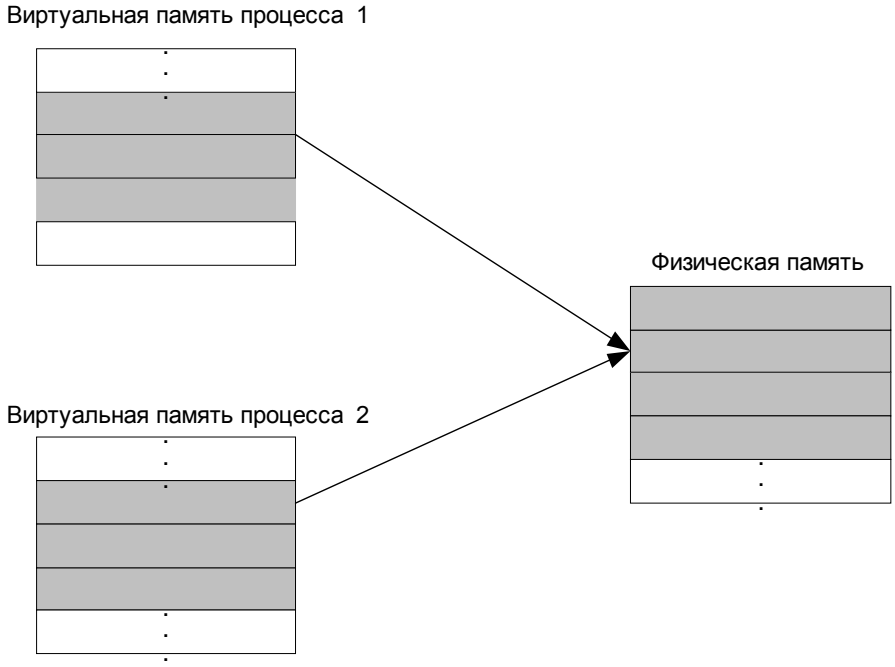
Виртуальная память представляет удобный механизм совместного использования памяти. Например, когда два процесса выполняют компиляцию программы на C, то можно экономить память, если загружать в нее только одну копию компилятора. Конечно, при этом каждому процессу нужна персональная память для хранения собственных кода и данных. Так как у каждого процесса есть отдельное адресное пространство, ОС достаточно загрузить компилятор один раз. Когда он будет вызван другим процессом, диспетчер виртуальной памяти просто отобразит виртуальные адреса этого процесса на страничные фреймы (физические адреса), в которых уже находится компилятор.

Аналогично, если два взаимодействующих процесса создают совместно используемый буфер памяти, то виртуальное адресное пространство каждого будет отображено на одни и те же страничные фреймы, занятые буфером. В случае с компилятором диспетчер виртуальной памяти не позволяет ни одному из процессов изменять страницы, занятые компилятором. В обоих процессах соответствующие виртуальные страницы помечаются признаком "только для чтения". А в случае с буфером возможность записи в него может потребоваться потокам обоих процессов, поэтому страницы помечаются признаком "чтение/запись". Но при таком совместном использовании необходимо синхронизировать доступ потоков к совместно используемой памяти, чтобы предотвратить одновременный их доступ и повреждение даных.

Секции, проекции и проецируемые файлы

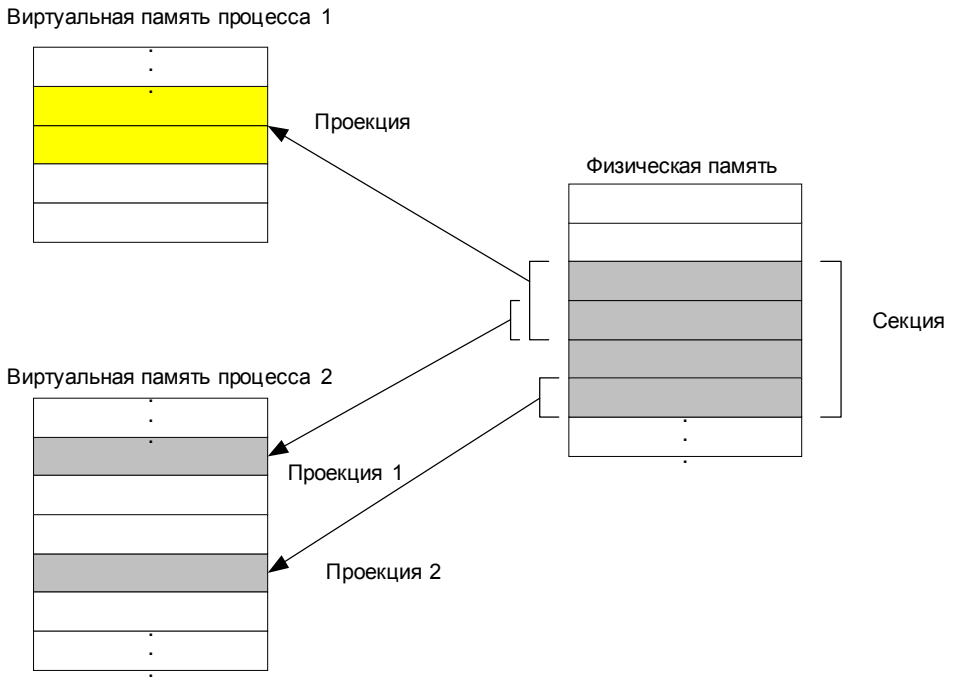
Совместно используемую память (***shared memory***) можно определить как память, видимую более чем одним процессом и присутствующую в нескольких виртуальных адресных пространствах. Подход Windows к совместному использованию ресурсов состоит в том, что они реализованы в виде защищенных объектов, и память не является исключением. ***Объект-секция*** (***section object***), представляет собой блок памяти, совместно используемый двумя или более процессами. Подсистема Win32 делает его доступным для своих клиентов как ***объект-проекцию файла*** (***file mapping object***). Поток одного из процессов создает объект-секцию и присваивает ему имя, чтобы потоки других процессов могли открыть его описатели. Открыв описатель объекта-секции, поток может отобразить секцию целиком или частично в виртуальное адресное пространство, свое собственное или другого процесса (см. *Рисунок 7.1*).

Рисунок 7.1 Совместное использование памяти



Объект-секция может быть большим и занимать много страниц. Для экономии своего виртуального адресного пространства процесс может отобразить только нужную ему часть секции; эта отображенная часть называется ***проекцией*** (***view***) ***секции***. Проекция служит окном в совместно используемую область памяти, и разные процессы могут отображать разные проекции секции или даже несколько проекций (см.*Рисунок 7.2*).

Рисунок 7.2 Отображение проекций секции



Проекции секции позволяют процессу работать с большими блоками памяти, для отображения которых целиком у него может не хватить виртуального адресного пространства. Например, пусть есть большая база данных с информацией о сотрудниках фирмы. Программа базы данных может создать объект-секцию, содержащую эту базу данных целиком. При обработке запроса к базе данных программа отображает проекцию секции с базой данных, считывает данные в свое адресное пространство, убирает отображение проекции, после чего отображает новую проекцию секции. Программа как бы перемещает "окно" по этой большой секции, считывая данные из всех частей базы и не переполняя свое виртуальное адресное пространство.

Содержимое совместно используемой памяти также откачивается на диск в файл подкачки, когда физической памяти становится недостаточно. Однако объект-секция может откачиваться и в ***проецируемый в память файл*** (***memory mapped file***). Примером такого файла является база данных сотрудников, а объект-секция используется для переноса содержимого файла в виртуальную память. В этом случае можно работать с файлом как с большим массивом, отображая различные проекции секции и производя чтение и запись в память, а не в файл. Этот процесс называется ***проекционным файловым вводом/выводом*** (***mapped file I/O***). Когда происходит обращение к недействительной странице, которой нет в физической памяти, происходит страничная ошибка, и диспетчер виртуальной

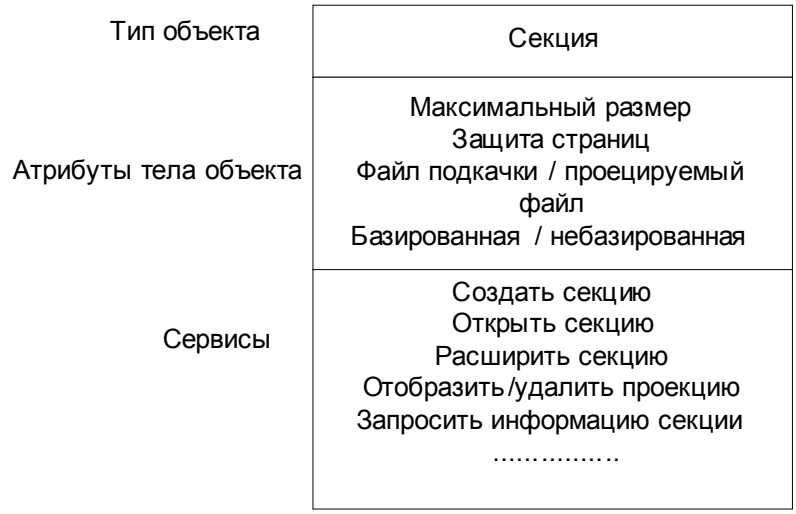
памяти перемещает эту страницу из проецируемого файла в память. Если приложение модифицировало страницу, то диспетчер виртуальной памяти записывает изменения на диске в процессе откачки страниц.

Для приложений Win32 проецируемые в память файлы служат удобным способом прямого доступа к большим файлам. Приложение создает для файла объект-проекцию файла (соответствующий объекту-секции), после чего выполняет чтение и запись по произвольным смещениям внутри файла. Диспетчер виртуальной памяти подкачивает нужные порции файла и записывает изменения обратно на диск.

Объект-секция

Как и в случае других объектов, диспетчер объектов создает и инициализирует заголовок объекта-секции. Диспетчер виртуальной памяти определяет тело объекта-секции и предоставляет сервисы, которые могут вызываться потоками пользовательского режима для чтения и изменения атрибутов, хранящихся в теле объекта-секции (см. *Рисунок 7.3*).

Рисунок 7.3 Объект секция



Максимальный размер - максимальное число байт, до которого может расти размер секции, а в случае проецируемого файла равен размеру файла.

Защита страниц - постраничная защита памяти назначается всем страницам секции при ее создании.

Файл подкачки / проецируемый файл - указывает создана ли секция пустой (резервное хранилище - файл подкачки) или загружена из файла (резервное хранилище - проецируемый файл).

Базированная / небазированная - секция должна располагаться по одному и тому же виртуальному адресу во всех использующих ее процессах или в разных процессах она может располагаться по разным виртуальным адресам.

Отображение проекции секции делает часть секции видимой в виртуальном адресном пространстве некоторого процесса. Аналогично, удаление проекции секции удаляет ее из адресного пространства процесса.

Совместное использование имеет место, когда два процесса отображают в свои адресные пространства фрагменты одной и той же секции. В этом случае процессы должны синхронизировать доступ к секции, чтобы избежать одновременного изменения данных. Для синхронизации можно использовать любые объекты синхронизации.

Для того чтобы отобразить проекцию секции, процесс должен сначала получить ее описатель. У процесса, создавшего секцию, этот описатель уже есть. Другие процессы могут получить этот описатель, используя именование, наследование или дублирование.

Защита памяти

В Windows используются четыре формы защиты памяти, первые три из которых используются в большинстве современных ОС:

- Отдельное адресное пространство для каждого процесса. Поток не может обращаться по виртуальным адресам другого процесса;
- Два режима работы:
  - ***режим ядра***, в котором потоки имеют доступ к системному коду и данным
  - ***пользовательский режим***, в котором потоки такого доступа не имеют
- Механизм постраничной защиты. С каждой виртуальной страницей связан набор флажков, определяющий, какие типы доступа к ней разрешены в пользовательском режиме и в режиме ядра;

- Объектная защита памяти. Этот механизм уникален только для Windows. Каждый раз, когда процесс открывает описатель объекта-секции или отображает проекцию секции, производится проверка, имеет ли процесс, пытающий выполнить операцию, соответствующие права доступа.

Память процесса

Когда поток обращается по виртуальному адресу, диспетчер виртуальной памяти исполнительной системы и аппаратура транслируют этот адрес в физический адрес. Система виртуальной памяти гарантирует, что потоки одного процесса не получат доступа к страничному фрейму, принадлежащему другому процессу.

Кроме защиты за счет трансляции виртуального адреса в физический, каждый процессор, обеспечивает аппаратную защиту памяти. Аппаратная защита минимальна, и ее требуется дополнить механизмами, предоставляемыми программным обеспечением виртуальной памяти. Из-за этого диспетчер виртуальной памяти Windows сильнее зависит от аппаратных особенностей, чем другие части системы.

Аппаратная страничная защита действует при всяком обращении потока к памяти. Каждая страница виртуальной памяти - это либо страница *пользовательского режима* (нижние 2 Гбайт), либо страницей *режима ядра* (верхние 2 Гбайт). Она имеет признак "только чтение" либо "чтение/запись".

Если поток исполняется в режиме ядра, то процессор позволяет ему читать любую действительную страницу памяти и изменять действительные страницы с признаком "чтение/запись". В пользовательском режиме поток может обращаться только к действительным страницам пользовательского режима и модифицировать те из них, которые имеют признак "чтение/запись". Процессор генерирует страничную ошибку при обращении к *недействительной* (отсутствующей в памяти) странице.

При попытке чтения или записи *действительной* страницы с нарушением доступа генерирует исключение нарушение доступа. Аппаратура может контролировать доступ только для *действительных* страниц (которые присутствуют в памяти). Если поток обращается к *недействительной* странице (той, которой в памяти нет) генерируется страничная ошибка и диспетчер виртуальной памяти подкачивает нужную страницу.

Диспетчер виртуальной памяти обеспечивает те же виды защиты, которые процессор предоставляет для действительных страниц:

- только чтение;
- чтение /запись;
- только исполнение (если есть аппаратная поддержка);
- нет доступа;
- копирование при записи.

Используя базовые сервисы виртуальной памяти, подсистема среды может управлять защитой личных страниц. Управление защитой страниц может повысить надежность программ, гарантируя, что потоки не модифицируют страницы "только для чтения".

Тип защиты "копирование при записи"

Тип защиты страниц "копирование при записи" - это средство оптимизации, которое диспетчер виртуальной памяти использует для экономии памяти. Если два процесса желают читать и изменять содержимое одной и той же области памяти, диспетчер виртуальной памяти назначает этой области защиту "копирование при записи". Затем оба процесса совместно используют эту память до тех пор, пока ни один из них не пишет в нее. Если поток одного из процессов модифицирует страницу этой области, диспетчер виртуальной памяти:

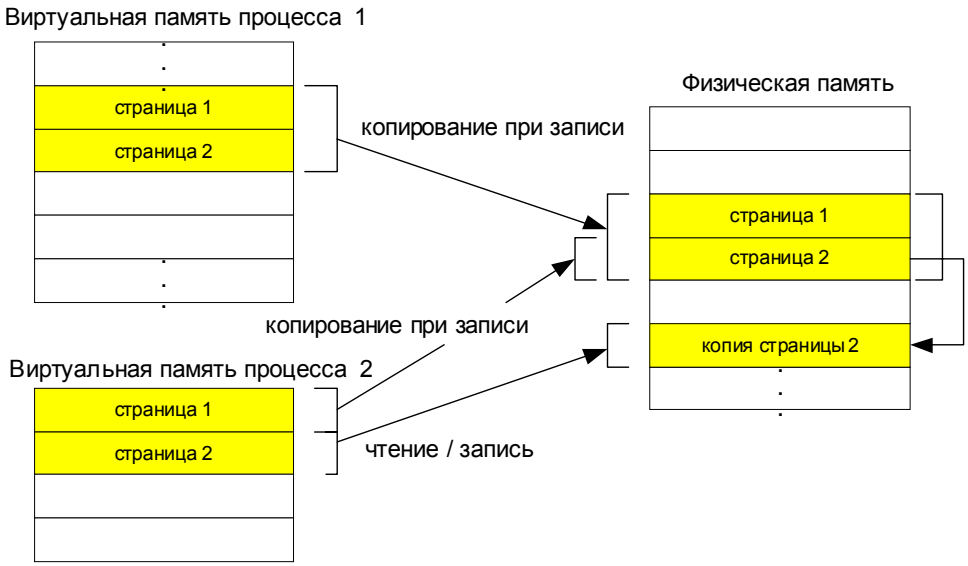
- копирует физический страничный фрейм в другое место памяти;
- модифицирует виртуальное адресное пространство процесса, чтобы оно указывало на копию;
- назначает новой странице защиту "чтение/запись".

Как показано на рисунке (см. *Рисунок 7.4*), скопированная страница невидима потокам других процессов. Таким образом, поток может изменять свою копию страницы, не затрагивая других использующих ее процессов.

Защита "копирование при записи" важна для страниц, содержащих код, она гарантирует, что изменения исполняемого образа затронет только тот процесс, чей поток выполнил это изменение. Например, первоначально страницы кода помечаются как "только исполнение". Однако, если в процессе отладки, программист задает точку останова, то отладчик должен поместить в код соответствующую машинную команду. Для этого отладчик сначала изменяет защиту страницы на

"копирование при записи". Диспетчер виртуальной памяти немедленно создает отдельную копию страницы для процесса, чей поток устанавливает точку останова. Остальные процессы продолжают использовать неизмененный вариант кода. Подсистема Win32 не позволяет приложениям непосредственно использовать защиту "копирование при записи".

Рисунок 7.4 Защита копирование при записи



Защита страниц "копирование при записи" - это один из примеров применения способа оптимизации, называемого ***отложенными вычислениями*** (***lazy evalution***), который используется диспетчером виртуальной памяти везде, где это возможно. Алгоритмы отложенных вычислений стараются не выполнять дорогостоящих операций до тех пор, пока это не станет абсолютно необходимым. Если операция так никогда и не потребуется, то на нее и не будет потрачено время. Например, в POSIX процесс вызывает функцию **`fork()`** для создания другого процесса. При этом ОС копирует все содержимое адресного пространства процесса создателя (родителя) в адресное пространство создаваемого процесса, а это требует больших затрат времени. Часто новый процесс сразу же вызывает функцию **`exec()`**, которая заново инициализирует его адресное пространство соответствующей исполняемой программой, что делает предыдущую операцию копирования напрасной. При использовании алгоритма отложенных вычислений диспетчер виртуальной памяти просто назначает страницам адресного пространства родительского процесса защиту "копирование при записи" и обеспечивает их совместное использование родителем и ребенком. Если дочерний (или родительский) процесс никогда не выполняет запись в свое адресное пространство, то оба

процесса продолжают использовать страницы совместно, и копирование не производится. Если же один из них все-таки выполняет запись, то диспетчер виртуальной памяти копирует только измененные страницы, а не целиком все адресное пространство.

Использование проецируемых в память файлов

Операции с файлами приходится делать практически во всех программах, и всегда это вызывает массу проблем. В Windows многие проблемы можно решить с помощью ***проецируемых в память файлов*** (***memory mapped files***).

Как и в случае с виртуальной памятью, проецируемые файлы позволяют резервировать регион адресного пространства и передавать ему физическую память. Различие между этими двумя механизмами состоит в том, что в последнем случае физическая память не выделяется из страничного файла, а берется из файла на диске. Как только этот файл спроецирован в память, к нему можно обращаться так, как будто он целиком в нее загружен.

Проецируемые файлы применяются для:

- загрузки и выполнения EXE и DLL файлов. Это позволяет существенно экономить как на размере страничного файла, так и на времени необходимом для подготовки приложения к выполнению;
- доступа к файлу данных на диске. Это позволяет обойтись без операций файлового ввода/вывода и буферизации его содержимого;
- разделение данных между несколькими процессами на одной машине.

Проецирование в память EXE и DLL файлов

При вызове функции **`CreateProcess`** система действует так:

1. Отыскивает EXE файл, указанный при вызове **`CreateProcess`**. Если файл не найден, то новый процесс не создается, а функция возвращает **`FALSE`**;
2. Создает новый объект процесс;
3. Создает адресное пространство нового процесса;

4. Резервирует регион адресного пространства - такой, чтобы в него поместился данный EXE файл. Желательное расположение этого региона указывается внутри самого EXE файл. При создании исполняемого файла приложения базовый адрес может быть задан через параметр компановщика /BASE;

5. Отмечает, что с рарезервированным регионом связан EXE файл на диске, а не страничный файл.

Спроецировав EXE файл на адресное пространство процесса, система обращается к разделу EXE файла со списком DLL, содержащим необходимые программе функции. После этого система, вызывая **`LoadLibrary`**, поочередно загружает указанные DLL модули. При этом система выполняет следующие действия:

1. Резервирует регион адресного пространства - такой, чтобы в него поместился заданный DLL файл. Желательное расположение этого региона указывается внутри самого DLL файла. При компановки DLL это значение можно изменить с помощью параметра /BASE;

2. Если зарезервировать регион по желательному для DLL базоваму адресу не удается, система пытается найти другой регион;

3. Отмечает, что с рарезервированным регионом связан DLL файл на диске, а не страничный файл.

Файлы данных, проецируемые в память

ОС позволяет проецировать на адресное пространство процесса и файл данных. Для этого нужно выполнить три операции:

1. Создать или открыть объект файл, идентифицирующий дисковый файл, который вы хотите использовать как проецируемый в память;

2. Создать объект проекция файла, чтобы сообщить системе размер файла и способ доступа к нему;

3. Указать системе, как спроецировать в адресное пространство вашего процесса объект проекция файла - целиком или частично.

Закончив работу с проецируемым в память файлом, следует выполнить тоже три операции:

1. Сообщить системе об отмене проецирования на адресное пространство процесса объекта проекция файла;

2. Закрыть этот объект;

3. Закрыть объект файл.

ЛАБОРАТОРНАЯ РАБОТА 7

ОТОБРАЖАЕМЫЕ В ПАМЯТЬ ФАЙЛЫ

Теория

Потоки процесса совместно используют его адресное пространство. При этом Windows обеспечивает защиту адресных пространств различных процессов. Поток одного процесса не может просто прочитать или записать в чужом адресном пространстве. Обеспечение механизма, который позволит совместно использовать информацию это классическая проблема для ОС. Мы уже рассматривали различные механизмы IPC. Можно использовать функции **`ReadProcessMemory`** и **`WriteProcessMemory`**, для того чтобы процессы могли прочитать или записать в адресные пространства друг друга. Однако отображаемые в память файлы это наиболее эффективный механизм для разделения информации между адресными пространствами процессов на одном компьютере. Windows спроектирована так, что отображаемые в память файлы, реализованы в рамках исполнительной системы, и, отвечает за реализацию, ***диспетчер виртуальной памяти*** (***Virtual Memory Manager***).

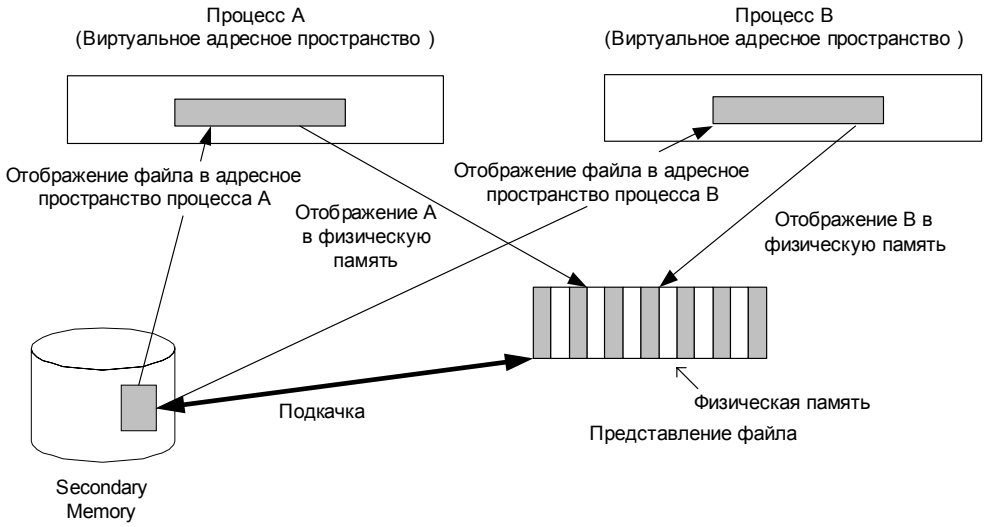
Концептуально, отображаемые в память файлы это дисковые файлы, которые загружаются в первичную память. Отображаемые в память файлы реализованы так же как виртуальная память Windows. В случае виртуальной памяти 4 Гб виртуальное адресное пространство определяется в страничном файле. Диспетчер виртуальной памяти (Virtual Memory Manager, VMM) загружает выбранные страницы из страничного файла в первичную память по запросу. Предположим, что тот же самый механизм использует обычный файл вместо страничного файла. Файл определяется как адресное пространство - поток байтов, который ассоциируется с порцией виртуального адресного пространства. Процесс может читать и писать в файл, просто читая или записывая по виртуальным адресам, в которые байты файла отображаются. При этом отображаемый в память файл подкачивается в первичную память из вторичной памяти без всяких усилий приложения (этим занимается VMM). Например, если процесс **A** открыл 64K файл и отобразил его в свое виртуальное адресное пространство с адреса 0x2000000 по 0x2000FFFF, тогда он может читать или записывать первый байт в файле, читая или записывая по адресу 0x2000000 или он может ссылаться на 16 байт используя адрес 0x2000000F.

Отображаемые в память файлы используются для загрузки EXE и DLL файлов. После того как процесс создан со своим виртуальным адресом пространством в 4 Гб, система проверяет размер EXE файла и резервирует столько памяти в виртуальном адресном пространстве процесса (с адреса 0x00400000). После этого, система отмечает, что вторичная память, которая соответствует этой части виртуального адресного пространства это EXE файл, а не страничный файл. Далее определяются DLL библиотеки, которые известны на период загрузки и резервируется виртуальное адресное пространство для каждой из этих DLL. И эта часть виртуального адресного пространства связывается с этими библиотеками, а не со страничным файлом.

Совместное использование отображаемых в память файлов

Существует еще один аспект использования отображаемых в память файлов, который делает их еще более полезными (*Рисунок 7.5*).

Рисунок 7.5 Отображаемые в память файлы



Мы можем получить доступ к информации, используя имя файла, причем несколько процессов могут отобразить один и тот же файл в свои виртуальные адресные пространства одновременно. Например, пусть процесс **A** отображает 64 К файл в свое адресное пространство с адреса 0x2000000 по адрес 0x2000FFFF. А процесс **B** открывает тот же самый файл и отображает его в свое виртуальное адресное пространство с адреса 0x3000000 по адрес 0x3000FFFF. Затем процесс **B** сможет "передать" информацию процессу **A**, записывая информацию по виртуальному адресу 0x30001234, а процесс **A** сможет прочесть ее по адресу 0x20001234. Windows гарантирует, что оба процесса будут видеть изменения в памяти при записи в нее.

Отображаемые в память файлы не имеют специального механизма, который позволяет управлять критической секцией.
Если отображаемый в память файл используется набором потоков и любой из потоков может обновить файл, то все потоки должны использовать события, мьютексы или семафоры для гарантии, что критическая секция используется эксклюзивно. Эти механизмы для защиты критической секции можно использовать для потоков одного процесса или для различных процессов, применяя наследование, дублирование и т.д.

Функции отображаемых в память файлов

Чтобы использовать отображаемые в память файлы вы должны:

- получить описатель файла, создав или открыв его;
- зарезервировать виртуальные адреса для файла;
- отобразить файл в виртуальное адресное пространство.

Этап 1: Создание или открытие объекта файл

Описатель файла можно получить, используя **CreateFile** или **OpenFile**. Для этого нужно вызвать функцию **CreateFile**:

```
HANDLE CreateFile(
LPCTSTR lpFileName, // указатель на имя файла
DWORD dwDesiredAccess, // режим доступа
DWORD dwShareMode, // режим совместного доступа
LPSECURITY_ATTRIBUTES lpSecurityAttributes,
// атрибуты безопасности
DWORD dwCreationDesposition, // как создать файл
DWORD dwFlagsAndAttributes, // атрибуты файла
HANDLE hTemplate
// описатель файла с атрибутами для копирования
);
```

LpFileName задает имя создаваемого или открываемого файла (при необходимости полное путевое имя).

DwDesireAccess - параметр, который может принимать только три значения:

- **0** (***no acces***s - нет доступа);
- **GENERIC_READ** (чтение);
- **GENERIC_WRITE** (запись).

Можно использовать их комбинации (**OR**). Создавая или открывая файл данных с намерением использовать его в качестве проецируемого в память, можно установить либо **GENERIC_READ** (только для чтения), либо комбинированный флаг **GENERIC_READ | GENERIC_WRITE** (чтение/запись).

Третий параметр, **DwShareMode**, указывает тип совместного доступа к данному файлу и может принимать следующие значения:

- **0** - использовать совместно нельзя;
- **FILE_SHARE_DELETE** - операции открыть будут успешны только, если они требуют доступ - удаление;
- **FILE_SHARE_READ** - операции открыть будут успешны только, если они требуют доступ - чтение;
- **FILE_SHARE_WRITE** - операции открыть будут успешны только, если они требуют доступ - запись.

Можно использовать их комбинации (**OR**).

Атрибуты безопасности трактуются также как у любого объекта.

DwCreationDesposition определяет, что должно происходить, если в момент создания (вызова **CreateFile**) файла он уже существует:

- **CREATE_NEW** - если файл существует, то будет ошибка;
- **CREATE_ALWAYS** - переписать существующий файл;
- **OPEN_EXISTING** - открыть существующий файл.

DwFlagsAndAttributes используется для передачи различных опций File Manager, в простейшем случае можно использовать **FILE_ATTRIBUTE_NORMAL**.

Htemplate можно использовать для передачи описателя другого файла, который содержит расширенные атрибуты.

Создав или открыв указанный файл, **CreateFile** возвращает его описатель, а в случае неудачи **INVALID_HANDLE_VALUE**.

Этап 2: Создание объекта проекция файла

Вызвав **CreateFile**, вы указали ОС, где находится память для проекции файла:

- на жестком диске;
- в сети;

- на CD-ROM.

Теперь необходимо сообщить системе, какой объем физической памяти нужен проекции файла. Для этого надо использовать функцию **CreateFileMapping**:

```
HANDLE CreateFileMapping(
HANDLE hFile,
// описатель файла, проецируемого на адресное пространство процесса
PSECURITY_ATTRIBUTES psa, // атрибуты безопасности
DWORD fdwProtect,// желательные атрибуты защиты
DWORD dwMaximumSizeHigh, // старшие 32 байта
DWORD dwMaximumSizeLow, // младшие 32 байта
PCSTR pszName); // имя объекта проекция файла
```

Создание файла, проецируемого в память, аналогично резервированию региона адресного пространства с последующей передачей ему физической памяти. Разница лишь в том, что физическая память для проецируемого файла - сам файл на диске, и для него не нужно выделять пространство в страничном файле. При создании объекта проекция файла система не резервирует регион адресного пространства и не увязывает его с физической памятью из файла (об этом дальше). Но как только дело дойдет до этого, системе понадобится знать атрибут защиты, присваиваемый страницам физической памяти.

Параметр **hfile -** это описатель файла, который возвращает функция **CreateFile (OpenFile).**

Параметр **psa** - это стандартные атрибуты безопасности.

Параметр **flProtect** принимает значение:

- **PAGE_READONLY** - отобразив объект проекцию файла на адресное пространство процесса, можно считывать данные из файла (в **CreateFile** должно быть задано **GENERIC_READ**);
- **PAGE_READWRITE** - отобразив объект проекцию файла на адресное пространство процесса, можно считывать данные из файла и записывать их (в **CreateFile** должно быть задано **GENERIC_READ | GENERIC_WRITE**);
- **PAGE_WRITECOPY** - отобразив объект проекцию файла на адресное пространство процесса, можно считывать данные из файла и записывать их. Запись приведет к созданию закрытой копии страницы (в **CreateFile** должно быть задано **GENERIC_READ** либо **GENERIC_READ | GENERIC_WRITE**).

Повторим еще раз, что значение этого параметра должны быть совместимы с привилегиями описателя файла. Например, если файл открыт только для чтения, то это означает, что вызывающий процесс сможет требовать только доступ - чтения региона, который содержит страницы файла.

Следующие два параметра этой функции **dwMaximumSizeHigh** и **dwMaximumSizeLow** самые важные. Максимальный размер отображаемой памяти это 64-битное значение и его две части передаются, используя эти два параметра. Основное назначение **CreateFileMapping** - гарантировать, что объекту проекция файла доступен нужный объем физической памяти. Так мы сообщаем системе максимальный размер файла в байтах. Чтобы создать объект проекцию файла таким, чтобы он соответствовал размеру файла, нужно задать оба параметра равными 0.

Параметр **lpName** - это имя объекта. Если объект не надо использовать совместно, то можно создать его безымянным (**NULL**).

Система создает объект проекцию файла и возвращает его описатель. Если объект создать не удалось, возвращается нулевой описатель (**NULL**).

Описатель, который возвращает функция **CreateFileMapping** можно использовать как любой другой описатель. Его можно передавать по наследству любому порождаемому (дочернему) процессу или он может быть передан в адресное пространство другого процесса, используя **DuplicateHandle**.

Можно использовать функцию **OpenFileMapping**:

```
HANDLE OpenFileMapping(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   LPCSTR lpName
);
```

Этап 3: Проецирование файловых данных на адресное пространство процесса

Когда объект проекция файла создан, нужно, чтобы система, зарезервировав регион адресного пространства под данные файла, передала их как физическую память, отображенную на регион. После того как объект создан, и адресное пространство выделено, необходимо отобразить содержимое файла в адресное пространство процесса. Это делает функция **MapViewOfFile**:

```
PVOID MapViewOfFile(
HANDLE hFileMappingObject, // описатель объекта проекция файла
DWORD dwDesiredAccess, // вид доступа к данным
DWORD dwFileOffsetHigh, //смещение в файле (старшие 32 бита)
DWORD dwFileOffsetlow, // смещение в файле (младшие 32 бита)
SIZE_T dwNumberOfBytesToMap // сколько байтов файла должно быть //
спроектировано (размер представления)
);
```

Параметр **hFileMappingObject** - это описатель объекта проекции файла, который возвращает функция **CreateFileMapping**.

Параметр **dwDesiredAccess** определяет, как можно использовать объект для доступа к данным. Он должен быть совместим с параметром **flProtect**, который используется в **CreateFileMapping**. Параметр **dwDesiredAccess** может принимать следующие значения:

- **FILE_MAP_WRITE**. Объект можно использовать для чтения и записи.
- **FILE_MAP_READ**. Потоки могут только читать из отображаемого в память файла, используя объект.
- **FILE_MAP_ALL_ACCESS**. Аналогично **FILE_MAP_WRITE**.
- **FILE_MAP_COPY**. Это значение задает "копирование при записи". Если объект создан с **PAGE_WRITECOPY**, а проецирование выполнено используя **FILE_MAP_COPY**, то процесс будет иметь проекцию файла, но при записи в файл исходный файл не изменится.

Вы можете отобразить весь файл в адресное пространство или только его часть, которая называется проекцией. 64-битное смещение (два параметра **DWORD**) **dwFileOffsetHigh** и **dwFileOffsetlow** положение указателя с которого начинается проекция, а **dwNumberOfBytesToMap** задает число байт в проекции. Указатель файла должен быть кратен ***единице резервирования памяти*** (***allocation granularity***). Функция выбирает подходящее место в адресном пространстве, в которое она будет отображать файл и возвращает это значение. Вы можете вручную выбрать местоположение, используя **MapViewOfFileE**x, которая имеет шестой параметр **LPVOID lpBaseAddress**. Этот параметр может быть установлен равным адресу, с которого начинается отображение. Он должен быть кратен единице резервирования памяти (*allocation granularity*).

Этап 4: Отключение файла данных от адресного пространства процесса

Когда необходимость в данных файла (спроецированного на регион адресного пространства процесса) отпадает, необходимо освободить регион вызовом:

```
BOOL UnmapViewOfFile(PVOID pvBaseAddress);
```

Параметр **pvBaseAddress** указывает базовый адрес региона, возвращаемый **MapViewOfFile**. Если вы не вызовите эту функцию, то регион не освободится до завершения вашего процесса.

Именованные каналы (Named Pipes)

Краткое описание ***именованных каналов*** приводится здесь, потому что их можно использовать при выполнении задания. ***Именованный канал*** - это IPC-механизм, который можно использовать для передачи данных между адресными пространствами процессов. Этот механизм *(именованные каналы)* специально спроектирован для того, чтобы обеспечить взаимодействие сервером с несколькими клиентами в сети. Именованные каналы отличаются от обычных каналов.

- Именованный канал может иметь несколько экземпляров. Все экземпляры имеют одинаковые параметры, так как они являются копиями одного и того же канала. Каждый экземпляр может быть использован различными парами процессов. Например, сервер может использовать именованный канал для создания класса именованных каналов, в этом случае каждый клиент, который соединяется с сервером, может использовать новый экземпляр именованного канала.

- Именованные каналы являются дуплексными. Это означает, что процесс может читать и писать, используя оба конца канала.

- Именованные каналы можно использовать в сети.

Коммуникационная связь по именованному каналу включает сервер именованного канала, и клиент именованного канала. Сервером именованного канала является приложение, создающее именованный канал, к которому подключаются клиенты.

Формат имени канала выглядит так: \\Сервер\Pipe\ИмяКанала. Элемент ***Сервер*** указывает компьютер, на котором работает сервер именованного канала (сервер не может создать именованный канал на удаленной машине), и его имя может быть DSN-именем (например, mspress.microsoft.com), NetBIOS-именем (mspress) или IP-адресом (255.0.0.0). Элемент Pipe должен быть строкой "Pipe", а ***ИмяКанала***-уникальное имя, назначенное именованному каналу.

Кроме того, при создании канала можно задать должен ли он быть двусторонним или односторонним, определить максимальное число одновременных соединений по данному каналу, и режим работы канала (побайтовой передачи или передачи сообщений).

Для создания именованного канала сервер использует функцию **CreateNamedPipe**:

```
HANDLE CreateNamedPipe(
     LPCTSTR lpName, // указатель на имя канала
     DWORD dwOpenMode, // режим открытия канала
     DWORD dwPipeMode, // режимы канала
     DWORD nMaxInstances, // максимальное количество экземпляров
     DWORD nOutBufferSize, // размер выходного буфера в байтах
     DWORD nInBufferSize, // размер входного буфера в байтах
     DWORD nDefaultTimeOut, // таймаут в миллисекундах
     LPSECURITY_ATTRIBUTES lpSecurityAttributes
     // указатель на структуру атрибутов безопасности
          );
```

Параметр **lpName** - это имя канала. Оно должно быть задано в форме \\.\pipe*pipename*, где \\.\- псевдоним локального компьютера.

Параметр **dwOpenMode** определяет несколько аспектов поведение канала, включая доступ к каналу, режим записи и безопасность.

- **PIPE_ACCESS_DUPLEX**. Информацию можно передавать по каналу в обоих направлениях.
- **PIPE_ACCESS_INBOUND**. Данные можно передавать по каналу только от клиента к серверу.
- **PIPE_ACCESS_OUTBOUND**. Данные можно передавать по каналу только от сервера к клиенту.

Если флаг **FILE_FLAG_WRITE_THROUGH**, тогда при записи в именованный канал по сети, функция не вернет управление до тех пор, пока информация не будет размещена в буфере на получающей машине. Флаги безопасности задают, как можно изменять установки защиты.

Параметр **dwPipeMode** определяет тип передаваемой информации, режимы чтения и ожидания:

- Если канал надо использовать для передачи потока байтов, то значение типа должно быть **PIPE_TYPE_BYTE**.
- Если канал надо использовать для передачи сообщений, то значение типа должно быть **PIPE_TYPE_MESSAGE**.
- Режим чтения можно задать и для потоков и для сообщений соответственно:
  - **PIPE_READMODE_BYTE**
  - **PIPE_READMODE_MESSAGE**
- Режим ожидания задает, блокироваться или нет:

- **PIPE_WAIT**
- **PIPE_NOWAIT**

Параметр **nMaxInstances** задает максимальное количество экземпляров, которые могут быть открыты для этого канала.

Параметры **nOutBufferSize** и **nInBufferSize** задают размеры для входного и выходного буферов канала соответственно.

Параметр **nDefaultTimeOut** задает таймаут по умолчанию (он может быть использован функцией **WaitNamedPipe**).

Параметр **lpSecurityAttributes** позволяет запретить несанкционированный доступ к именованному каналу.

Чему нужно научиться?

Нужно научиться использовать возможности проекционного файлового ввода/вывода (mapped file I/O).

Задания

Уровень 1 (A)

Подготовить исходный файл с содержимом (например, *abcdefghij*…). Используя отображаемый в память файл, поработать с исходным файлом, читая и записывая в память. Убедиться в том, что исходное содержимое файла изменилось.

Скелет кода для 1(A)

```
#include <stdio.h>
#include <windows.h>

int main(int argc,char* argv[])
{
  // Открыть исходный файл
HANDLE hfile=CreateFile(…);
  if(hfile==INVALID_HANDLE_VALUE)
  {
    printf("Cann't create file! %d\n",GetLastError());
    exit(0);
  }

  HANDLE hfilemap=CreateFileMapping(…);
  if(hfilemap==NULL)
  {
    printf("Cann't create file mapping! %d\n",GetLastError());
    exit(0);
  }
```

```
  // Получить размер файла dwfilesize

  CloseHandle(…);

  PVOID pvfile=MapViewOfFile(…);

  CloseHandle(…);

  PSTR pchANSI=(PSTR)pvfile;
  for(int i=0;i<dwfilesize;i++)
  {
    printf("ch=%c\n",pchANSI[i]);
    pchANSI[i]='t';
  }

  UnmapViewOfFile(…);

  return(0);

}
```

Уровень 2, 3 (А)

Задача подобна той, которую мы уже решали, но в данном случае более сложная. Процесс производитель (источник) и процесс потребитель (приемник) читают и записывают в обычные файлы. Также существуют еще два процесса шифровальщик и дешифровальщик. Они должны фильтровать информацию, передаваемую производителем и принимаемую потребителем.

На рисунке (*Рисунок 7.5*) показано, как эти четыре процесса должны взаимодействовать друг с другом. Производитель читает поток байтов из обычного файла и пишет в первый отображаемый в память файл. Шифровальщик читает отображаемый в память файл, шифрует каждый байт в потоке и записывает в именованный канал. Дешифровальщик читает зашифрованные байты из канала, дешифрует их и записывает во второй, отображаемый в память файл.

Потребитель читает данные из второго отображаемого в память файла и записывает их во второй обычный файл.

Ваш алгоритм шифрования может быть очень простым, например, преобразовывать маленькие буквы в большие и наоборот. Алгоритм дешифрования должен делать все наоборот.

Необходимо, чтобы процессы работали с разными скоростями.

Скелеты кода для 2 и 3 (А)

Главная программа

```
// Главная программа, которая создает все процессы и позволяет им
// (производителю, шифровальщику, дешифровальщику и потребителю)
// совместно использовать информацию
int main(int argc, char *argv[])
{
// создать канал для передачи информации между процессами
// шифровальщика и дешифровальщика
// создать процесс производитель
  if(!CreateProcess(…))
  {
    fprintf(stderr, "…", GetLastError());
    getc(stdin);
    ExitProcess(1);
  }
  Sleep(500); // Дать возможность производителю начать работать

// создать процесс шифровальщик
  if(!CreateProcess(…))
  {
    fprintf(stderr, "…", GetLastError());
    getc(stdin);
    ExitProcess(1);
  }
  Sleep(500); // Дать возможность шифровальщику начать работать

// создать процесс дешифровальщик
  if(!CreateProcess(…))
  {
    fprintf(stderr, "…", GetLastError());
    getc(stdin);
    ExitProcess(1);
  }
  Sleep(500); // Дать возможность дешифровальщику начать работать

// создать процесс потребитель
  if(!CreateProcess(…))
  {
    fprintf(stderr, "…", GetLastError());
    getc(stdin);
    ExitProcess(1);
  }
  Sleep(500); // Дать возможность потребителю начать работать
// Подождать пока производитель, шифровальщик, дешифровальщик
// и потребитель закончатся
}
```

Производитель

```
// Процесс-источник читает информацию из файла-источника, а затем использует
// отображаемый в память файл для передачи информации процессу шифровальщику

int main(int argc, char *argv[])
{
  // открыть файл источник
  sourceFile = CreateFile(…);
  if(sourceFile == INVALID_HANDLE_VALUE)
  {
  fprintf(stderr, GetLastError());
    getc(stdin);
    ExitProcess(1);
  }
  // открыть отображаемый в память файл
  peMMFFile = CreateFile(…);
  if(peMMFFile == INVALID_HANDLE_VALUE)
  {
  fprintf(stderr, GetLastError());
    getc(stdin);
    ExitProcess(1);
  }
  // Создать объект проекцию файла
  peMMFFile = CreateFileMapping(…);
  if(peMMFMap == NULL)
  {
  fprintf(stderr, GetLastError());
    getc(stdin);
    ExitProcess(1);
  }

// Спроецировать (отобразить)
  baseAddr = (PBYTE) MapViewOfFile(…);
  if(baseAddr == NULL)
  {
  fprintf(stderr, GetLastError());
    getc(stdin);
    ExitProcess(1);
  }

  srand(P_RAND_SEED); // Set random seed

  // главный цикл обработки файла источника
  while(…)
  {
    // Использовать синхронизацию для защиты критической секции
Sleep(rand()%timeToProduce);
    // Запись информации в отображаемый память файл
// за счет записи по адресу baseAddr
  }
// Закончить
}
```

Индекс

А

Б

В

Д

З

И

К

Л

М

Н

О

П

P

C

T

У

Ф

Ч

Э

Я

B

C

D

E

H

I

K

L

N

O

P

S

T

V

W

Учебно - образовательный комплекс

ФТК

Операционные системы
Часть II. UNIX

Санкт-Петербургский Государственный Политехнический Университет
***Ф**акультет **Т**ехнической **К**ибернетики*

Санкт-Петербург, Политехническая ул. д.29
www.spbstu.ru

СОДЕРЖАНИЕ

Введение

История

В мире ОС прогресс достигается медленно. Разработка ОС занимает В 1965 г. в Bell Telephone Laboratories (BTL) начали разрабатывать новую операционную систему MULTiplexed Information and Computing Service (MULTICS). Хотели создать многопользовательскую (сотни пользователей) и многозадачную операционную систему разделения времени. Кен Томпсон и Денис Ритчи участвовали в разработке от BTL. Система MULTICS так и не была завершена и в 1969 году BTL вышла из проекта. Используя наработки в 1969 году, операционная система все-таки создали. Она включала:

- файловую систему;
- подсистему управления процессами;
- набор утилит.

Система была написана на ассемблере для PDP-7 и назвали ее UNIX (uni - один) (передразнивая MULICS multi - много).

Когда, в 1971 году патентному отделу BTL понадобилась система обработки текста, в качестве ОС выбрали UNIX. Систему перенесли на PDP-11:

- система занимала 16 К;
- для прикладных программ можно было использовать 8 К;
- максимальный размер файла, который система поддерживала был 64 К;
- максимальный размер дискового пространства, который система поддерживала, был 512 К.

После создания первых ассемблерных версий Томпсон начал работать над компилятором для FORTRAN и разработал язык B (интерпретатор), а Ритчи переработал его в язык C. В 1973 году ядро ОС было переписано на языке высокого уровня C. Теперь появилась возможность переносить ОС UNIX на другие аппаратные платформы. В соответствии с федеральным законом BTL не имела права коммерческого распространения UNIX, но начиная с 1974 года начала продавать ее университетам по символической цене для образовательных целей. ОС модернизировалась, и каждая версия снабжалась соответствующей

редакцией руководства программиста. Всего было выпущено 10 версий редакций. Первые 7 редакций были разработаны в BTL (*Таблица В.1*). Их разработкой занималась Computer Research Group (CRG). Они были предназначены для PDP-11, а потом для VAX. UNIX System Group отвечала за сопровождение системы.

Таблица В.1 Основные редакции

1-я редакция	1971	написана на ассемблере и включает компилятор B
3-я редакция	1973	включает компилятор C
4-я редакция	1973	ядро написано на C
6-я редакция	1975	вышла за пределы BTL (BSD-появляется)
7-я редакция	1979	включает командный интерпретатор Боурна

В 1989 году появилась System V Release 4 (SVR4), которая объединила возможности нескольких версий UNIX:

- предыдущих версий System V;
- Sun OS фирмы Microsystems;
- BSD UNIX компании Berkley Software Distribution.

Она включала:

- Командные интерпретаторы Корна и C (BSD);
- Символические ссылки;
- Отображаемые в память файлы (Sun OS);
- Сетевую файловую систему NFS и RPC (Sun OS);
- Программный интерфейс сокетов (BSD).

BSD

Калифорнийский университет в Беркли получил одну из первых лицензий на операционную систему UNIX в декабре 1974 года. За несколько лет группа выпускников университета, в состав которой входил Билл Джой (Bill Joy) и Чак Хелей (Chuck Haley), разработала несколько утилит для этой системы, в том числе редактор vi и компилятор языка Паскаль. Все созданные приложения были собраны в единый пакет под названием Berkeley Software Distribution (BSD) и продавалась весной 1978 года по цене $50 за одну лицензию.

Коммерческие версии

Популярность UNIX привела к тому, что различные компании начали выпуск и продажу своих собственных вариантов системы. Они использовали одну из базовых реализаций UNIX от BTL или Беркли, переносили ее на другие машины и включали дополнительные возможности.

В 1982 году Билл Джой покинул Беркли и стал одним из создателей корпорации Sun Microsystems, которая выпустила свой вариант UNIX на основе 4.2 BSD SunOS. В дальнейшем компанией была разработана еще одна версия системы, Solaris на основе 4.3 BSD.

Компании Microsoft и BTL совместно создали систему XENIX. Позже SCO перенесла SVR 3 на платформу 80386 и выпустила эту систему под названием SCO UNIX.

В 80-х годах появилось множество различных коммерческих вариантов системы UNIX:

- AIX от IBM;
- HP-UX от Hewlett-Packard Corporation;
- ULTRIX (выпущенной вслед за DEC OSF/1, переименованной позже в Digital UNIX) от компании Digital.

Стандарты

Распространенность различных реализаций UNIX привела к появлению проблем ***совместимости***. Несмотря на то, что все существующие варианты на первый взгляд "похожи на UNIX", на самом деле они существенно различаются между собой. Существование отличий было заложено изначально, за счет наличия двух основных веток:

- системы BTL System V
- BSD (созданной в Беркли).

Появление коммерческих вариантов UNIX только усугубило проблему.

Системы System V и 4 BSD существенно отличались. Они имели различные несовместимые между собой файловые системы, реализации поддержки сетей и архитектуры виртуальной памяти. Некоторые различия были обусловлены дизайном ядра систем, но большинство из них находилось на уровне программирования интерфейса. Это

приводило к тому, что невозможно было создать приложение, которое работало бы без каких-либо изменений в обеих операционных системах.

Все коммерческие варианты UNIX строились на основе либо System V, либо BSD, к которым производители добавляли дополнительные возможности. Эти добавления часто оказывались непереносимыми на другие платформы. В результате создатели приложений вынуждены были тратить огромное количество времени и усилий, для того чтобы их программы нормально функционировали в различных реализациях UNIX.

Для решения проблемы необходимо было разработать некий стандартный набор интерфейсов, чем и занялись несколько групп энтузиастов. В результате появилось множество стандартов. Они отличались друг от друга, как и варианты UNIX.

В каждом из стандартов описывалось взаимодействие между программами и операционной системой, но не затрагивался вопрос реализации самого интерфейса взаимодействия. В них определялись наборы функций и подробно приводились их конструкции. Совместимые системы должны удовлетворять требованиям, изложенным в стандартах, однако реализация необходимых функций могла быть произведена как на уровне ядра, так и на уровне библиотек пользователя. Стандарты также определяли поднабор функций, предлагаемых большинством систем UNIX. Теоретически, если пользователь будет использовать при написании приложения только те функции, которые входят в этот набор, то созданное приложение будет переносимо на любую систему, совместимую со стандартами.

В 1986 году организация IEEE поручила специальному комитету разработать и опубликовать стандарты на среды операционных систем. Название POSIX (Portable Operating System based on UNIX) переводится как "Переносимые операционные системы, основанные на UNIX". Эти документы описывали компоненты ядра, систем SVR 3 и 4.3 BSD. Стандарт POSIX1003.1, более известный как POSIX.1, был опубликован в 19990 году. Многие производители приняли этот стандарт, так как он не ограничивался каким-то одним вариантом системы UNIX.

X/Open это международный консорциум производителей компьютерной техники и программного обеспечения. Он был сформирован в 1984 году. Его целью являлась не только разработка новых стандартов, но и создание открытой среды Common Applications Environment (Общей программной среды, CAE), базирующейся на уже существующих стандартах. Консорциум опубликовал семи томный труд "X/Open Portability Guide" (XPG), последнее (четвертое) издание

которого вышло в 1993 году. Материал руководства был основан на стандарте POSIX.1, расширял его и описывал многие дополнительные области, такие как интернационализация, оконные интерфейсы и обработка данных.

LINUX

MINIX - это демонстрационная ОС, написанная известным ученым компьютерщиком Эндрю Танненбаумом. Она предназначалась для демонстрации различных концепций ОС. Линусу было 23 года, он пытался создать более полную версию UNIX для пользователей MINIX. 5 октября 1991 года Линус представил миру первую официальную версию LINUX.

"Жалеете ли вы о добрых старых временах, когда был MINIX и когда мужчины- были мужчинами и сами писали драйверы устройств? Или у вас, нет подходящего проекта и вам до смерти хочется вонзить зубы в какую-нибудь ОС и переделать ее под свои задачи? И, нет уже фанатиков, готовых просиживать ноги, чтобы сделать великолепную программу? Тогда, я пишу именно для вас. Я работаю над бесплатной версией, подобной MINIX для компьютеров 386, работа на стадии, действующей, и я собираюсь организовать распространение исходных текстов".

Кому принадлежит LINUX

Авторские права на OS/2 принадлежит IBM, на MS DOS и Windows Microsoft. LINUX тоже не "бесправная система". Авторские права на разные ее части принадлежат разным людям. Линус Торвальде владеет правами на основное ядро Linux, Red Hat обладает правами на пакет Red Hat, Пол Фолькердинг на пакет Slackware. Многие утилиты распространяются по лицензии GNU General Public License (GPL). Copyright это авторское право, а copylef это авторское лево. Существует фонд бесплатного программного обеспечения (Free software Foundation). Его организовал Ричард Сталлмен. По лицензии GPL (авторское лево) применение, модификация и распространение ПО доступно любому и каждый может переделать его как хочет. Единственным условием является предоставление переделанного исходного текста. При этом создатель сохраняет свое авторское право, но он должен предоставить другим такие же возможности.

LINUX сегодня - это бесплатная, многозадачная и многопользовательская ОС. Вам не нужно обновлять свои программы каждые несколько лет за непомерные деньги. Для LINUX их можно получить бесплатно через Интернет. Кроме того, имея исходные тексты

можно модифицировать систему по своему вкусу. Если у вас на работе UNIX, то почему не иметь дома LINUX. Если вы администратор UNIX, то почему не брать работу на дом и не выполнять ее в LINUX. Кроме того, LINUX обеспечивает простой доступ в Internet.

Развитие системы

Система UNIX сильно изменилась за годы своего существования. Все начиналось с небольшой операционной среды, которую использовала группой людей в одной лаборатории. Сегодня UNIX - это одна из основных операционных систем. В настоящее время UNIX используется на самых различных системах, начиная от небольших встроенных контроллеров и заканчивая огромными мэйнфреймами. Под управлением UNIX работают различные приложения.

UNIX постоянно развивается потому, что перед системой встают все новые и новые задачи. А хороший дизайн системы позволяет добавлять в нее новые возможности по мере развития технологий. Создатели системы начали работу с построения простых, но расширяемых базовых средств. Несмотря на то, что UNIX является целостной операционной системой, это не значит, что она не будет изменяться дальше.

Функциональные возможности

Главной причиной, приводящей к изменениям, является необходимость добавления новых возможностей в систему. Изначально новые функциональные средства появились в UNIX только как пользовательские инструменты и утилиты. Позже, когда UNIX превратилась во вполне развитую систему, разработчики стали добавлять многие дополнительные возможности прямо в ядро ОС.

Многие из новых функций системы создавались для поддержки более сложных программ. Например, **Interprocess Communication (IPS)** в System V:

- разделяемая память,
- семафоры,
- очереди сообщений.

Эти средства позволяют процессам взаимодействовать, используя совместно данные, обмениваясь сообщениями и синхронизируя свои действия. Большинство современных систем UNIX поддерживают создание многопоточных приложений.

Возможности IPC и использование потоков помогают при разработке сложных приложений, основанных на модели "клиент-сервер". В таких программах сервер должен постоянно ожидать запроса от клиентов. Если запрос приходит, сервер обрабатывает его и снова переходит в режим ожидания. Будет еще лучше, если сервер сможет обслуживать сразу несколько клиентских запросов. Для этого сервер может создавать отдельный процесс для обработки каждого запроса, а все процессы могут совместно использовать одни и те же данные. Многопоточная система позволяет реализовать сервер как один процесс, имеющий несколько потоков, которые выполняются в одном адресном пространстве.

Частью любой операционной системы является файловая система. В ОС UNIX также было добавлено множество новых возможностей, в том числе поддержка файлов FIFO (First In, First Out), символьных связей, а также файлов, имеющих размеры большие, чем раздел диска. Современные системы поддерживают защиту файлов, списки прав доступа, а также ограничения доступа к дискам для каждого пользователя.

Сетевая поддержка

За годы развития UNIX больше всего изменилась сетевой подсистемой. Ранние версии ОС работали изолированно, однако распространение компьютерных сетей привело к необходимости их поддержки в системе UNIX. Сначала эту проблему начали решать в университете Беркли. Организация DARPA профинансировала проект по встраиванию поддержки TCP/IP в 4BSD. Сегодня системы UNIX поддерживают большое количество сетевых интерфейсов (таких как Ethernet и ATM), протоколов (TCP/IP и UDP/IP) и других средств (например, сокетов и STREAMS).

Появление возможности соединения с другими компьютерами сильно повлияло на операционную систему. Пользователи ОС сразу же захотели совместно использовать файлы, расположенные на соединенных между собой машинах, а также запускать приложения на удаленных узлах. Для удовлетворения этих требований развитие системы UNIX велось в трех направлениях:

- Разрабатывались распределенные файловые системы, позволявшие вести прозрачный доступ к файлам на удаленных узлах (например, Network File System, NFS);
- Создавалось большое количество распределенных служб, позволяющих совместно использовать информацию по сети. (например, Network Information Service, NIS);

- Появлялись распределенные операционные системы.

Производительность

Разработчики постоянно стремятся увеличить производительность ОС. Конкурирующие поставщики операционных систем стремятся продемонстрировать, что именно их ОС самая производительная. С этой целью вносятся изменения во все внутренние подсистемы.

ОС должна максимально использовать возможности современных аппаратных технологий. Это означает необходимость ее переноса на новые и более высокопроизводительные процессоры. После того как ядро UNIX было почти полностью переписано на языке C, решение этой задачи стало относительно легким. В прошлом разработчики тратили огромные усилия на отделение от общего кода программы участков, написанных для конкретного аппаратного оборудования, и включение их в отдельные модули. После проведения этих действий для переноса системы требовалось переписывать заново только отдельные модули. Обычно такие модули отвечали за обработку прерываний, трансляцию виртуальных адресов, переключение контекстов и драйверы устройств.

Кроме того, в современных условиях UNIX должна поддерживать несколько процессоров. Изначально ядро UNIX разрабатывалось для одного процессора, поэтому защита структур данных при параллельном доступе к ним одновременно нескольких процессоров не была предусмотрена. Позже сразу несколько производителей разработали многопроцессорные реализации системы UNIX. В большинстве из них использовалось традиционное ядро UNIX, к которому добавляли элементы защиты, обеспечивающие безопасность общих структур данных.

В 70-е годы производительность систем UNIX ограничивалась скоростью работы процессора и размером памяти. Но вскоре стали использоваться технологии:

- свопинга;

- страничной организации памяти.

По мере развития вычислительной техники скорость доступа к памяти и процессору стала играть меньшую роль. Система большую часть времени занималась переносом страниц между дисками и оперативной памятью. Это привело к появлению новых разработок в области файловых систем и систем виртуальной памяти.

Качество

Все преимущества ничего не значат, если в системе есть ошибки. Разработчики вносили множество изменений в систему, чтобы сделать ее более устойчивой и надежной.

Используемый вначале механизм оповещения был ненадежен и неэффективен. Позже его реализация была пересмотрена и появилась новая, более надежная система оповещения - надежные сигналы.

В различных системах UNIX предлагается стандартная утилита fsck, которая проверяет и восстанавливает поврежденные файловые системы.

Изменения в компьютерном мире

Появилась новая модель вычислений клиент/сервер, в которой компьютеры, называемые серверами, предоставляют различные службы рабочим станциям, (клиентам):

- Для хранения файлов пользователей стали применяться файловые серверы.
- Серверы приложений - это компьютеры, оснащенные одним или несколькими мощными процессорами, на которых пользователи могут выполнять задачи, требующие большого объема вычислений (например, решения математических уравнений).
- На серверах баз данных выполняется специальная программа, обрабатывающая запросы к базам данных, поступающие от клиентов.

Серверы - это мощные высокопроизводительные машины с быстродействующими процессорами и большими объемами оперативной и дисковой памяти. Клиентские рабочие станции имеют меньшую производительность, размеры памяти и объемы дисков, но они, оснащаются высококачественными мониторами и предоставляют пользователю большие интерактивные возможности.

Постепенно рабочие станции становились все более производительными и различия между клиентами и серверами стирались. Кроме того, централизованное выполнение необходимых служб на небольшом количестве серверов приводило к перегрузкам сетей и серверов. В результате подход к построению компьютерных систем изменился. Появилась новая технология распределенных вычислений. Например, каждый узел сети может выступать одновременно в качестве клиента и сервера. Он может предоставить

собственную файловую систему для удаленного доступа с других узлов, а кроме того, выступать как клиент по отношению к файлам, хранящимся на других узлах. Такой подход уменьшает перегрузки сети.

Поддержка приложений

Система UNIX изначально разрабатывалась для применения в исследовательских лабораториях и университетах. Система позволяла некоторому количеству пользователей выполнять программы обработки и редактирования текстов и математических вычислений. После того как UNIX стала популярной, ею стали пользоваться для решения более широкого круга задач. К началу 90-х годов UNIX использовалась в физических и космических лабораториях, мультимедийных рабочих станциях для обработки звука и видео, а также во встроенных контроллерах, использующихся в различных системах.

Модели

Одним из преимуществ дизайна системы UNIX были:

- ее небольшой размер;
- простота;
- ограниченный набор основных функций.

Главной целью разработчиков системы было предоставить простые инструменты, которые можно комбинировать между собой, используя например ***каналы*** (***pipe***). Традиционное ядро было монолитным и его расширение было трудной задачей. Чем больше функций добавлялось в ядро, тем оно все больше разрасталось и усложнялось. От первоначального размера, не превышающего 100 Кбайт, оно достигло объема нескольких мегабайтов.

Очень хотелось получить расширяемую и модульную ОС. Наиболее удачной реализацией такой системы стала Mach. Она послужила основой для коммерческих ОС (OSF/1 и NextStep). Система Mach использовала архитектуру микроядра, в которой небольшое ядро предоставляет средство для выполнения ограниченного набора основных функций и занимается передачей сообщений, а серверные процессы на пользовательском уровне предоставляют все остальные функции.

Микроядро не может быть таким же производительным, как традиционное монолитное ядро. Существуют системы со страничной поддержкой и динамической загрузкой компонентов ядра, которые позволяют загружать или выгружать из памяти компоненты ядра по мере необходимости.

Преимущества UNIX

Рассмотрим причины популярности UNIX.

- **Способ распространения системы.** Корпорация BTL продавала лицензии и исходные коды системы по достаточно низкой цене. Поэтому UNIX стала популярной среди многих пользователей по всему миру. Так как в комплект поставки входили исходные коды, пользователи имели возможность экспериментировать с ними, улучшать их, а так же обмениваться друг с другом созданными изменениями. Корпорация BTL включала многие нововведения в следующие версии системы. Разработчики из Беркли действовали также. Учебные заведения, коммерческие организации и хакеры-энтузиасты из разных стран все принимали участие в развитии системы. Это был открытый процесс. Многие производители коммерческих вариантов UNIX поддержали концепцию открытых систем и тоже сделали свои разработки доступными.

- **Хороший дизайн UNIX.** Красота в краткости. Ядро небольшого размера имело минимальный набор основных служб. Кроме того, был разработан набор полезных утилит. Механизм конвейера совместно с ***оболочкой*** (***shell***) позволял пользователям комбинировать эти утилиты различными способами и создавать свои инструменты.

- **Унифицированный интерфейс с устройствами ввода-вывода.** Все устройства представлены как файлы. Система позволяет пользователям применять один и тот же набор системных вызовов для доступа и работы с различными устройствами, как с файлами. Разработчики могут не заботиться, с чем именно производится обмен, с файлом, терминалом, принтером или другим устройством.

- **Переносимость.** Большая часть ядра написана на языке C. Система, написанная на ассемблере, выигрывала в производительности на 20%, а ее объем был на 20-40% меньше от объема системы, переписанной на C. Использование C позволило переносить UNIX на новые аппаратные платформы. Первая реализация системы была для PDP-11 и затем была перенесена на VAX-11.

- **Многозадачная и многопользовательская система.**

- **Большое количество приложений, начиная от текстовых редакторов и заканчивая системами управления базами данных.**

Недостатки UNIX

- **Пользовательские приложения.** UNIX - это замечательная ОС, однако, большинству пользователей нужна не сама система, а определенные приложения. Пользователей не интересует структура файловой системы или модель вычислений. Они хотят работать с определенными программами (например, текстовыми редакторами, финансовыми пакетами или программами для создания изображений), потратив на это минимум усилий.
- **Графический интерфейс**. Пользователям нужен простой унифицированный графический интерфейс. В первых системах UNIX его не было и пользователям это не нравилось.

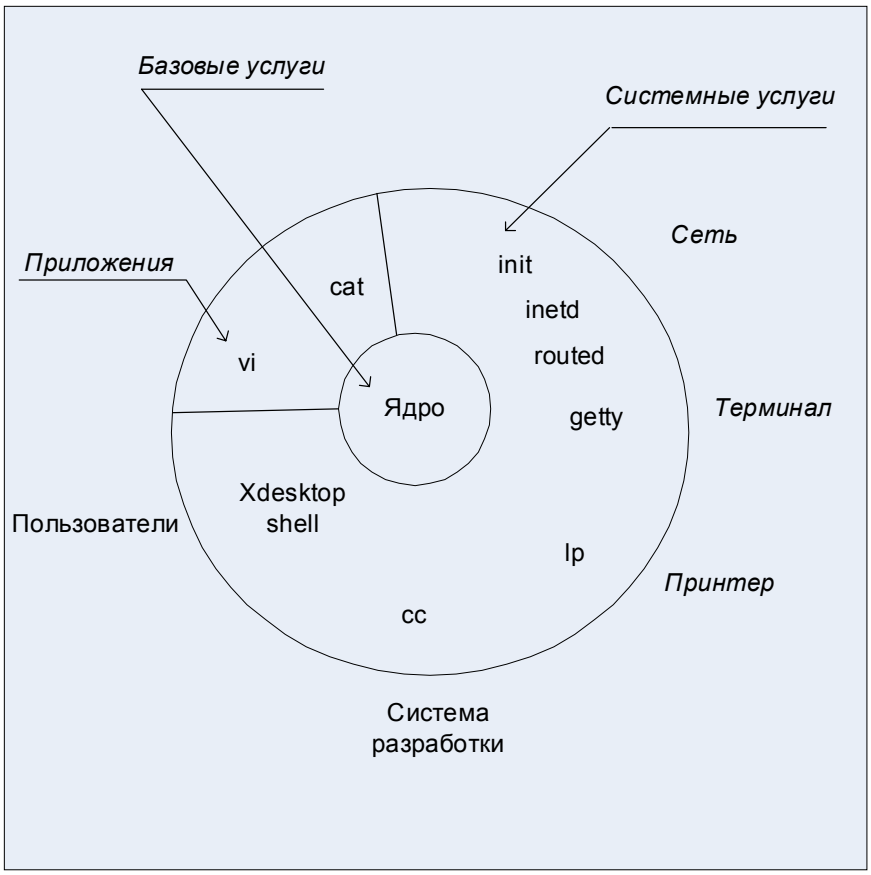
Как сказал Ритчи: "UNIX является простой и понятной системой, но чтобы понять и принять ее простоту, требуется гений (или, как минимум, программист)". Получилось так, что UNIX, требует от пользователей, желающих эффективно работать в системе, творческого мышления и определенной изобретательности. Однако большинство пользователей предпочитают простые в изучении интегрированные многофункциональные программы, подобные тем, что применяются на персональных компьютерах под управлением Windows.

Глава 1

Архитектура UNIX

Двухуровневая модель системы представлена на рисунке (*Рисунок 1.1*).

Рисунок 1.1 Модель системы UNIX



В центре находится ***ядро*** (***kernel***). Ядро непосредственно взаимодействует с аппаратной частью компьютера, изолируя прикладные программы от особенности ее архитектуры. Ядро предоставляет набор услуг прикладным программам. К услугам ядра относятся операции ввода/вывода (открытия, чтения, записи и

управления файлами), создания и управления процессами, их синхронизации и межпроцессорного взаимодействия. Все приложения запрашивают услуги ядра посредством системных вызовов.

Второй уровень составляют приложения или задачи, как системные, определяющие функциональность системы, так и прикладные, обеспечивающие пользовательский интерфейс UNIX. Несмотря на внешнюю разнородность приложений, схемы их взаимодействия с ядром одинаковы.

Далее рассмотрим отдельные компоненты ядра системы.

Ядро

Ядро системы обеспечивает базовую функциональность операционной системы:

- создает процессы и управляет ими;
- распределяет память;
- обеспечивает доступ к файлам и периферийным устройствам.

Взаимодействие прикладных задач с ядром происходит посредством стандартного интерфейса системных вызовов. Интерфейс системных вызовов представляет собой набор услуг ядра и определяет формат запросов на услуги. Процесс запрашивает услугу посредством системного вызова определенной процедуры ядра, похожего на обычный вызов библиотечной функции. Ядро от имени процесса выполняет запрос и возвращает процессу необходимые данные. Структура ядра представлена на рисунке (*Рисунок 1.2*).

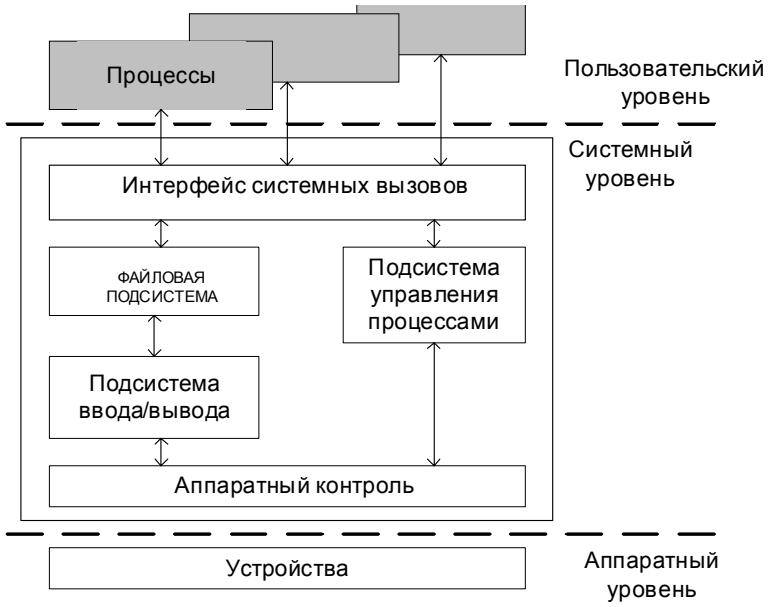
Ядро состоит из трех основных подсистем:

· Файловая подсистема;

· Подсистема управления процессами и памятью;

· Подсистема ввода/вывода.

Рисунок 1.2 Внутренняя структура ядра UNIX



Файловая подсистема

Файловая подсистема обеспечивает унифицированный интерфейс доступа к данным, расположенных на дисковых накопителях, и к периферийным устройствам. Одни и те же функции **`open(2)`**, **`read(2)`**, **`write(2)`** могут использоваться как при чтении или записи данных на диск, так и при выводе текста на принтер или терминал.

Файловая подсистема контролирует права доступа к файлу, выполняет операции размещения и удаления файла, а также выполняет запись/чтение данных файла. Поскольку большинство прикладных функций выполняется через интерфейс файловой системы (в том числе и доступ к периферийным устройствам), права доступа к файлам определяют привилегии пользователя в системе.

Файловая подсистема обеспечивает перенаправление запросов, адресованных периферийным устройствам, соответствующим модулям подсистемы ввода/вывода.

Подсистема управления процессами

Запущенная на выполнение программа порождает в системе один или более процессов (или задач). Подсистема управления процессами контролирует:

- Создание и удаление процессов;

- Распределение системных ресурсов (памяти, вычислительных ресурсов) между процессами;
- Синхронизацию процессов;
- Межпроцессорное взаимодействие.

Число активных процессов может превышать число процессоров компьютера, но в каждый конкретный момент времени на каждом процессоре может выполняться только один процесс. Операционная система управляет доступом процессов к вычислительным ресурсам, создавая иллюзию одновременного выполнения нескольких задач.

Специальный модуль ядра, называемый ***планировщиком*** процессов (***scheduler***), разрешает конфликты между процессами в конкуренции за системные ресурсы (процессор, память, устройства ввода/вывода). Планировщик запускает процесс на выполнение, следя за тем, чтобы процесс монопольно не захватил разделяемые системные ресурсы. Процесс освобождает процессор, ожидая длительной операции ввода/вывода, или спустя выделенный квант времени. В этом случае планировщик выбирает следующий процесс, имеющий наивысший приоритет, и запускает его на выполнение.

Модуль управления памятью обеспечивает размещение оперативной памяти для прикладных задач. Оперативная память является дорогим ресурсом, и ее не бывает "слишком много". При недостаточном количестве памяти для всех процессов, ядро перемещает части процессов во вторичную память (в специальную область жесткого диска), освобождая память для выполняющегося процесса. При этом реализуется виртуальная память: процесс выполняется в логическом адресном пространстве, превышающем доступную физическую память. Управление виртуальной памятью является задачей модуля управления памятью.

Модуль межпроцессорного взаимодействия отвечает за уведомление процессов о событиях и обеспечивает обмен данными между процессами.

Подсистема ввода/вывода

Подсистема ввода/вывода выполняет запросы файловой подсистемы и подсистемы управления процессами для доступа к периферийным устройствам (дискам, магнитным лентам, терминалам и т.д.). Она обеспечивает буферизацию данных и взаимодействие с *драйверами устройств* - специальными модулями ядра, обслуживающими внешние устройства.

Программирование в среде командного процессора (SHELL)

Несмотря на то, что в последние годы в системе UNIX стали применяться графические интерфейсы, большинство утилит для работы и управления системой вызываются из командной строки. Интерпретатор командной строки UNIX называется **оболочкой** (***shell***).

Сразу же после регистрации пользователя в системе UNIX делает для него текущим его рабочий каталог и запускает программу-оболочку. Оболочка - это программа для получения от пользователя команд и их выполнения (соответствуют файлу command.com в MS DOS).

В Red Hat LINUX доступны следующие оболочки: **sh**, **bash** (Bourne Again Shell), **tcsh**, **csh**, **pdksh** (Public Domain Korn Shell), **zsh**, **ash** и **mc**. Можно выбрать ту, которая вам больше понравится. **bash** является оболочкой по умолчанию в большинстве дистрибутивов LINUX, а **sh** доступна в большинстве UNIX систем.

Оболочка играет роль первичного интерфейса между пользователем и операционной системой, но сама не является частью ядра операционной системы.

Оболочка Bourne **sh** это настоящая оболочка UNIX. Она была создана Стивом Борном и Джоном Машея из AT&T Bell Laboratories. Выполняемая программа этой оболочки находится в файле **/bin/sh**. Она широко доступна, обладает мощными возможностями и широко используется.

Оболочка C, **csh** была разработана Биллом Джоем в Калифорнийском университете в Беркли. Выполняемая программа этой оболочки находится в файле **/bin/csh**. Оболочка Борна превосходит ее по возможностям программирования. Синтаксис команд оболочки C очень напоминает язык программирования C.

Чтобы определить какая из оболочек используется, введите **`echo $SHELL`**.

Оболочка, которая запускается сразу, после регистрации пользователя указывается в файле паролей (**etc/passwd**). Можно редактировать файл вручную, но перед редактированием лучше сохранить копию файла. Каждая запись файла **/etc/passwd** имеет семь разделенных двоеточиями полей:

`Login name`:**`Password`**:**`UserID`**:**`GroupID`**:**`Full Name`**:**`Home directory`**:**`Command environment`**(**`Shell`**)

Описание полей представлено в таблице (*Таблица 1.1*).

Таблица 1.1 Описание полей /etc/passwd

Поле	*Описание*
Login name	Имя для входа пользователя в систему. Оно должно быть уникальным в пределах файла
Password	Не сам пароль пользователя, а зашифрованный ключ к нему
UserID (UID)	Уникальный числовой идентификатор пользователя. Любой созданный пользователем файл ассоциируется с этим числовым идентификатором
GroupID (GID)	Уникальный числовой идентификатор группы, к которой принадлежит пользователь. Любой созданный пользователем файл ассоциируется с этим числовым идентификатором
Full Name	Поле комментария, например полное имя пользователя
Home directory	Домашний каталог пользователя. В начале сеанса работы система помещает пользователя в этот каталог (обычно **/Home/Login name**)
Command environment(Shell)	После завершения процесса входа в систему текст, находящийся в этом поле, выполняется как команда. Обычно эта команда запускает оболочку **(/bin/bash**)

Настройка рабочей среды

Прежде чем появляется приглашение оболочки, система устанавливает для пользователя его рабочую среду. Среда - это данные и установки, которые применяются во время работы пользователя в системе. Вы можете изменять эти установки по своему желанию.

Рабочая среда состоит из двух частей:

- ***Установки терминала*** (***Terminal environment***) - управляет настройками терминала;
- ***Окружение оболочки*** (***Shell environment***) - это некоторое количество переменных и их значений.

Рассмотрим сначала установки терминала. Получает и выполняет команды пользователя оболочка, но все, что вводит пользователь, сначала обрабатывается драйвером устройства. Драйвер может работать в двух режимах:

- ***С распознаванием*** управляющих клавиш (***cooked mode***) - каждая нажатая пользователем клавиша анализируется драйвером устройства;
- ***Без распознавания*** управляющих клавиш (***raw mode***) - все вводимые пользователем символы сразу передаются оболочке.

Наиболее важные управляющие символы:

- ***прервать*** (***interrupt***) прекращает выполнение текущей программы (обычно <**Del**> или <**Ctrl+C**>);
- ***удалить*** (***erase***) удаляет из буфера последний введенный пользователем символ (обычно <**Backspace**>);
- ***очистить*** (***kill***) полностью очищает буфер (обычно <**@**>);
- ***конец строки*** (***end-of-line***) сообщает, что пользователь закончил набирать команду и ее нужно анализировать (обычно <**Enter** >);
- ***конец файла*** (***end-of-file***) (обычно <**Cntr+D**>).

Для установки и отображения этих управляющих символов используется команда **`stty`** (***set teletype***).

```
stty
.......intr=^C; kill=^U; eof=^D
stty kill '^C'
```

Теперь рассмотрим окружение оболочки. Окружение складывается из типа используемой пользователем оболочки, рабочего каталога пользователя, типа используемого терминала и т.д. Многие из этих переменных устанавливаются во время регистрации в системе. Переменные окружения устанавливаются следующим образом: **`VARIABLE=value`**.

Перечислим наиболее важные переменные окружения:

- **`HOME=/home/login`** задает рабочий каталог пользователя;
- **`LOGNAME=login`** задает имя зарегистрировавшегося пользователя;
- **`PATH=path`** задает список каталогов, которые оболочка просматривает в поисках команды (например, **`PATH=/usr:/bin:/usr/bin`**);

- **`PS1=prompt`** задает первичное приглашение оболочки (***primary shell prompt***). Вы можете задать свое приглашение **`PS1`**="**`Enter your Command Please`**!";
- **`SHELL=shell`** задаеет местонахождение программу, указанной в качестве оболочки;
- **`TERM=termtype`** задает тип терминала (например, **`TERM=vt100`**);
- **`PWD=directory`** задает текущее положение в дереве каталогов.

Если вы пользуетесь **bash** и хотите, чтобы переменные окружения устанавливались всякий раз, когда вы регистрируетесь в системе, задайте их в конфигурационном файле **.profile**. Если вы применяете **C shell**, то необходимо использовать файл **.login**.

Добавить каталог в переменную **`PATH`** можно следующим образом: **`PATH=$PATH:newpath`**.

Процесс разбора командной строки оболочкой

Разбор (***parsing***) - процесс разделения того, что вводит пользователь на составные части для дальнейшей обработки. При этом обрабатываются универсальные символы в именах файлов, подставляются переменные оболочки, производится перенаправление ввода/вывода, подготавливаются группы команд, выполняется подстановка результатов выполнения команд. Только после этого введенная командная строка может быть выполнена.

Команды, флаги и параметры

Если вы хотите, чтобы команда правильно работала, ее необходимо правильно задать. Название или имя команды должно быть первым элементом в строке.

Имя команды

При выполнении команды ищется файл с таким именем. Например, мы хотим выполнить команду вывода содержимого каталога **`ls`**. Файл с таким именем находится в каталоге **/bin**. Если этот каталог задан в переменной **`PATH`**, то оболочка найдет и выполнит программу **/bin/ls**. Некоторые команды не являются независимыми файлами, а встроены в оболочку. Это - так называемые встроенные команды (например, команда смены каталога **`cd`**).

Флаги (опции)

За именем могут следовать ***флаги*** (***опции***). Флаги (опции) - это отдельные буквы, предваренные знаком дефиса и изменяющие поведение команды. Например, команда **`ls`** просто выводит содержимое текущего каталога, а **`ls -l`** выводит еще и атрибуты. Мы использовали флаг **`-l`**. Флаги можно объединять **`ls -lF`** или **`ls -l -F`**.

Параметры

Параметры - это последовательности символов, разделенные любым из символов, указанных в переменной окружения **`IFS`** ***(Inter Field Separator***). Текущими переменными-разделителями являются пробел, а также символы табуляции и конца строки. Между параметрами можно помещать любое число символов разделителей, при разборе оболочка сократит количество этих символов до одного.

```
command <пробел> <пробел> <пробел> <табуляция> parameter
command <пробел> parameter
ls -l ourscript
```

Задание имен файлов

* - любой набор символов, кроме * когда с нее начинается имя файла;

? - любой одиночный символ;

[ ] - символ из заданного диапазона;

```
*.txt - все файлы с расширением txt;
ls *rep*;
ls  .*rep*;
rm *;
rm * txt;
rm  -i *txt;
ls ??x; ls *.???; ls [A-Z]*;
```

Связывание процессов с помощью каналов (pipe)

Выход одной программы можно использовать в качестве входа для другой с помощью канала **|** - ***канал*** (***pipe***)

```
sort allsales | lp
cat sales* | sort | lp
```

Перенаправление ввода/вывода

В LINUX&UNIX ввод с клавиатуры соответствует чтению из файла **stdin**, вывод на терминал соответствует выводу в файл **stdout**. Можно перенаправить ввод и вывод так, чтобы вместо ввода с терминала или вывода на терминал информация считывалась бы из файла или перенаправлялась в файл:

- < - принять ввод из файла
- > - направить вывод в файл
- >> - добавить в конец файла

mail grcomerce < letter

`date > filename` (если файл **filename** не существует, то он будет создан)

Использование переменных оболочки

`HOME=/usr/home/login` (задает рабочий каталог пользователя)

`LOGNAME=login` (имя зарегистрировавшегося пользователя)

`PATH=path` (список каталогов, просматриваемых оболочкой в поисках команды (**`PATH=/usr:/bin:/usr/bin`**))

`PS1=prompt` (первичное приглашение оболочки (можем изменить **`PS1="Enter command:"`**))

`PWD=directory` (текущая директория)

`SHELL=shell` (например, /bin/bash)

`TERM=termtype`

`PATH=$PATH:newpath`

Подстановка результатов выполнения команд (command substitution)

Команда в обратных кавычках будет заменена на результат ее выполнения.

```
echo Today data and time are `date`
Today data and time are Mondey September 18 14:35:09
```

Группы команд, порожденные оболочки

Завершая команды, можно поместить несколько команд в одной командной строке, образовав, таким образом, группу команд. Оболочка будет выполнять команды последовательно, как если бы каждая из них была введена в отдельной строке:

```
command1; command2; command3
```

Если надо перенаправить ввод или вывод для нескольких команд сразу можно образовать группу команд, заключив их в фигурные скобки:

```
{command1; command2} > output_file
```

Вывод группы команд можно перенаправить в канал:

```
{command1; command2} | command3
```

Команды в группе выполняются текущей оболочкой. Если одна из команд модифицирует окружение или сменит каталог, изменения останутся в силе и после того, как выполнится вся группа. Этого можно избежать, если выполнить группу команд ***порожденной оболочкой*** (***subshell***).

```
(command1; command2) | command3
```

Сценарии оболочки (shell scripts)

Сценарий оболочки это несколько команд оболочки, помещенные в файл. Чтобы создать сценарий необходимо:

- Воспользоваться текстовым редактором, чтобы сохранить команды в файле;
- Сделать этот файл выполнимым: **`chmod +x script_name`**;

Необходимо проверить, что **`PATH`** содержит путь к этому файлу (**`echo $PATH`**) или выполнить из его директории (**./script_name**).

Чтобы писать программы, нужны переменные и управляющие структуры. Переменная в каждый момент времени может принимать одно из множества допустимых значений. Управляющие структуры дают возможность задать последовательность выполнения команд и делятся на структуры ветвления (**`if then else`**, **`case`**) и циклические структуры (**`for`**, **`while`**). С помощью ветвления в зависимости от значения переменной или результата выполнения команды, мы можем выбрать один из нескольких вариантов. Циклические структуры можно использовать для повторяющегося выполнения последовательности команд.

Использование переменных в сценариях

Присваивать переменным значения можно четырьмя способами:

- непосредственное присваивание;
- с помощью команды **`read`**;
- использование параметров командной строки;
- подстановка результата выполнения команды.

Непосредственное присваивание

```
Variable_name=Variable_value
Myemail=tvk@ics2.ecd.spbstu.ru, mydir=/usr/home/tvk

echo "My e-mail address is $myemail"
cp myfile $mydir
```

Команда read

Команда **`read`** приостанавливает сценарий и ожидает ввода с клавиатуры (со стандартного ввода). После нажатия **Enter** выполнение сценария продолжается.

```
echo "Enter file name"
read filename
cp $filename $mydir
```

Использование параметров командной строки

При разборе командной строки оболочка связывает с каждым параметром командной строки имя переменной. Параметрами командной строки являются последовательности символов, разделенные пробелами или табуляцией. Связанные с параметрами командной строки переменные называются ***позиционными параметрами*** (они соответствуют позициям в строке - **`$1 ... $9`**).

```
script param1 param2 … param9
              $1         $2              $9
```

Существует еще особый тип переменных - ***встроенные переменные*:**

$# - количество позиционных параметров;

$0 - имя программы (скрипта);

$* - строка, содержащая все аргументы командной строки;

$? - статус завершения команды (0 - успех);

```
script_test
Echo "Number of parameters is $#";
Echo "Program (script) name is $0";
Echo "Parameters as single string is $*";
```

Если параметров больше 9, то можно использовать **$*** и **shift**:

```
shift [количество] (по умолчанию 1).
```

Подстановка результата выполнения команды

```
mydir=`pwd`
echo $mydir
```

Использование управляющих структур

Операторы case, if then else

```
case str in
  образец) операторы;;
  образец) операторы;;
            *) операторы;;
esac

script_check_logname1

who | grep $1 > /dev/null
case $? In
echo "$1 is logged in";;
*) echo "$1 is not here! Try again later";;
esac

script_check_logname2

if
  who | grep $1 > /dev/null
then
  echo "$1 is logged in"
else
  echo "$1 is not here! Try again later"
fi
```

Сравнение выражений с помощью команды test

```
test выражение
  [выражение]
```

В качестве выражения можно использовать:

- строковое сравнение;
- сравнение чисел;
- файловые операторы;

· логические операторы.

Рассмотрим сначала строковое сравнение:

- **=** две строки равны;
- **!=** две строки не равны;
- **-n** длина строки больше нуля;
- **-z** длина строки равна нулю.

```
str1="abc"
str2="abd"
if[str1=str2]
then
   echo "str1 equal str2"
else
   echo "str1 not equal str2"
fi
if[-n str2]
then
   echo "str2 has length >0"
else
   echo "str2 has length equal to zero"
fi
if[-z str1]
then
   echo "str1 has length equal to zero"
else
   echo "str1 has length >0"
fi
```

Можно сравнивать числа:

- **-eq** два числа равны;
- **-ge** одно число больше либо равно другому;
- **-le** одно число меньше либо равно другому;
- **-ne** два числа не равны.

```
if[num1 -eq num2]
then
   echo "num1=num2"
else
   echo "num1!=num2"
fi
```

Можно использовать файловые операторы:

- **-f** файл существует и является обычным файлом;
- **-d** файл является каталогом;
- **-r** файл существует и доступен для чтения;
- **-w** файл существует и доступен для записи;
- **-x** файл существует и является выполнимым;
- **-z** файл пустой.

```
case $# in
  if test !-r $1     // файл читать нельзя!
     then
   exit(1)
     fi
     if test -f $2     // существует ли второй файл
     then
   if test -w $2     // разрешена ли запись во второй файл
     then
     echo "$2 exists, copy over it?(Y/N)"
     read response
     case response in
     Y/y) cp $1 $2;;
     *)     exit(1);;
     esac
   else
     exit(1) // не разрешена
   fi
    else
   cp $1 $2 // второго файла не существует
     fi
```

Можно использовать логические операторы:

- **!** отрицает логическое выражение;
- **&&** логическая операция **AND** для двух логических выражений;
- **||** логическая операция **OR** для двух логических выражений.

Операторы while, until, for

```
while выражение
  do
   операторы
  done
```

Выполнять пока выражение истинно (**true**).

```
until выражение
   do
    операторы
   done
```

Выполнять пока выражение ложно (**false**).

```
for текущее значение из списка in список
   do
    операторы
   done
```

Операторы будут выполнены по одному разу для каждого значения из списка.

```
for filename in `ls`
do
   cp $filename /backup/$filename
   if[$? -ne 0]
   then
     echo "copy for $filename failed!"
   fi
done
```

Экспорт переменных

Вновь созданные переменные оболочки и значения, присвоенные уже существующим, действуют в пределах выполняющейся оболочки.

```
today=Monday
echo $today
Monday
Whatday_script
echo "Today is $today"
today=Friday
echo "Today is $today"
chmod +x Whatday_script
today=Monday
Whatday_script
```

ЛАБОРАТОРНАЯ РАБОТА 1

ПРОГРАММИРОВАНИЕ НА SHELL

Теория

Программирование в среде командного процессора (**Shell**).

Чему нужно научиться ?

Необходимо научиться писать программы на shell.

Задания

Уровень 1, 2, 3 (A)

Придумайте сами или реализуйте одну из следующих программ на shell:

- "безопасное" копирование файлов;
- выявление процессов-"захватчиков" системы;
- уничтожение "вредных" процессов;
- автоматический секретарь (будильник);
- поиск "хулиганов" диска;
- идентификация входа пользователя в систему;
- аудит состояния процесса;
- аудит домашних каталогов;
- поиск заданного файла с минимальным маршрутом;
- поиск заданного файла с максимальным маршрутом;
- частотный анализатор текста.

Глава

2

Процессы и нити

Процессы в любой ОС играют ключевую роль. От эффективной работы системы управления процессами зависит производительность системы в целом. Ядро ОС предоставляет задачам базовый набор услуг, определяемый интерфейсом системных вызовов. Мы рассмотрим управление процессами. Дополнительные функциональные возможности системы, т.е. услуги, которые она предоставляет пользователям, также определяются активными процессами. От того, какие процессы выполняются в вашей системе, зависит, является ли она сервером базы данных или, например, вычислительным сервером.

Программой называют совокупность файлов (исходные файлы, объектные файлы или исполняемый файл). Для того, чтобы программа могла быть запущена на выполнение, ОС должна сначала создать окружение или среду выполнения (сюда входят память, системные ресурсы, включая услуги ядра). Это окружение получило название процесса. Мы можем представить процесс как совокупность данных ядра системы, необходимых для описания образа программы в памяти и управления ее выполнением. ***Процесс*** - это программа в стадии выполнения и все выполняющиеся в UNIX программы представлены в виде процессов.

Ядро

Ядро - это программа, которая управляет аппаратурой, создает, уничтожает все процессы и управляет ими. Ядро управляет процессами и предоставляет им различные услуги.

Ядро UNIX состоит из двух секций:

- ***Программной секции***;
- ***Секции управляющих таблиц***.

Из программной секции нас будет интересовать ***система управления процессами***. А из секции управляющих таблиц мы рассмотрим ***таблицу процессов***.

Задача ядра, как диспетчера ресурсов, оптимально распределять ресурсы системы:

- память
- процессор (CPU)
- и т.д.

Процесс, которому не доступен необходимый ресурс, приостанавливается до тех пор, пока ресурс не станет доступен. Процессор является таким же ресурсом. Ядро системы предоставляет процессам промежуток времени, называемый ***квантом времени***, в течение которого они имеют доступ к процессору (обычно около 10 мс) По окончании кванта времени процессор передаются другому процессу. Такая модель функционирования получила название ***квантование времени*** (***time-slicing***).

Система UNIX функционирует следующим образом:

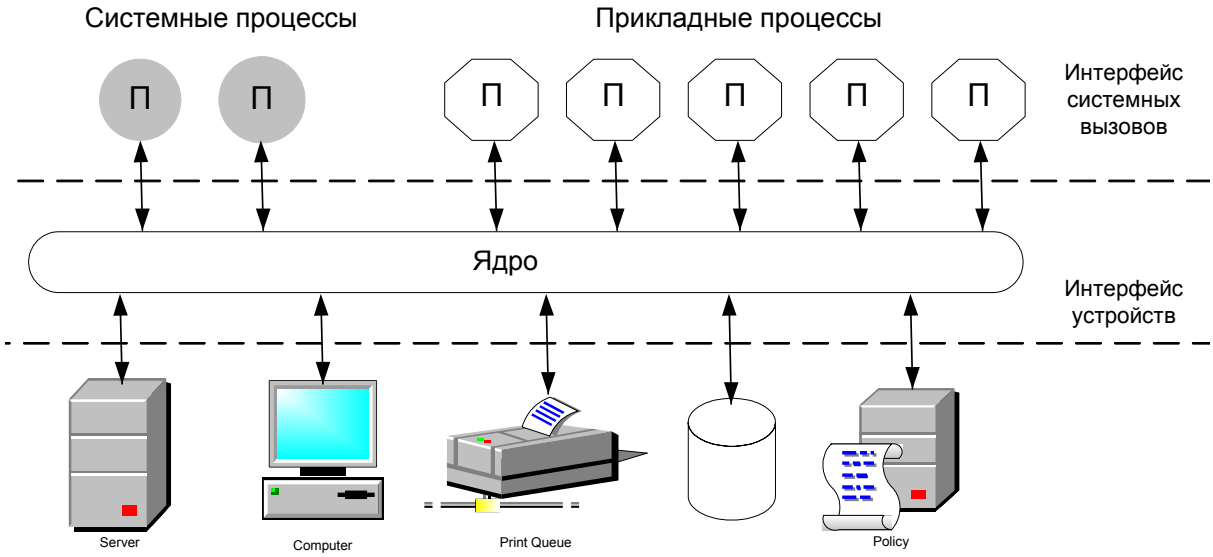
- Прикладные процессы запрашивают услуги ядра при помощи ***системных вызовов*** (см. *Рисунок 2.1*), являющегося центральным компонентом API системы UNIX. Ядро выполняет эти запросы от имени вызывающих процессов.
- При некорректных действиях процесса (например, попытка деления на ноль или переполнения стека) возникают ***аппаратные исключения***. Их обработка требует вмешательства ядра, а потом происходит их обработка от имени процесса.
- Ядро обрабатывает ***аппаратные прерывания*** от устройств. Ядро воспринимает прерывания как глобальные события не относящиеся к какому-либо процессу.

Режимы ядра и задачи

Прикладные программы работают в *режиме задачи*, а функции ядра выполняются в *режиме ядра*. Ядро защищает часть адресного пространства от доступа в режиме задачи. Более того, наиболее привилегированные машинные инструкции могут выполняться только в режиме ядра. Главной причиной появления различных режимов выполнения является безопасность. Если пользовательские процессы выполняются в менее привилегированном режиме, то они не могут

случайно или специально повредить другой процесс или ядро системы. Ошибки в программе имеют локальный характер и обычно не влияют на выполнение других процессов.

Рисунок 2.1 Взаимодействие ядра с процессами и устройствами



Большинство реализаций UNIX используют *виртуальную память*. В системе виртуальной памяти, адреса выделенные программе, не ссылаются непосредственно на физическое размещение в памяти. Каждому процессу предоставляется собственное *виртуальное адресное пространство*, а виртуальные адреса *транслируются* в физические при помощи *карт трансляции адресов*.

Часть виртуального адресного пространства отображается на код и структуры данных ядра. Эта часть называется ***системным пространством*** или ***пространством ядра.*** В системе может одновременно выполняться только одна копия ядра, следовательно, все процессы ссылаются на единое адресное пространство ядра. В ядре содержатся глобальные структуры данных ОС.

Мы уже говорили о том, что процессы не могут напрямую обращаться к ядру и должны использовать для этого ***интерфейс системных вызовов***. После того как процесс производит ***системный вызов***, запускается специальная последовательность команд (называемая ***переключателем режимов***), переводящая систему в режим ядра, а управление передается ядру, которое обрабатывает операцию от имени процесса. После завершения обработки системного вызова ядро

вызывает другую последовательность команд, возвращающую систему обратно в режим задачи (производится еще одно переключение режима), и передает управление текущему процессу.

Рассмотрим что такое *контекста выполнения* (Рисунок 2.2). Функции ядра могут выполняться:

- *в контексте процесса*
- *в системном контексте.*

В контексте процесса ядро функционирует от имени текущего процесса. Ядро также должно выполнять глобальные задачи, такие как обслуживание прерываний устройств и пересчет приоритетов процессов. Такие задачи не зависят от какого- либо конкретного процесса и, следовательно, обрабатываются в системном контексте (также называемом ***контекстом прерываний***).

Рисунок 2.2 Взаимодействие ядра с процессами и устройствами

Выполнение в режиме ядра

Существует три типа событий, которые переводят систему в режим ядра:

- ***прерывания устройств*** (***interrupts***);
- ***исключения*** (***exceptions***);
- а также ***ловушки*** (***traps***) или ***программные прерывания*** (***software interrupts***).

Прерывания - это асинхронные события, происходящие в периферийных устройствах: диски, терминалы, аппаратные таймеры. Так как прерывания не зависят от текущего процесса, они должны обрабатываться в системном контексте, при этом доступ в адресное пространство им не требуется. ***Исключения*** возникают по ходу работы процесса по причинам, зависящим от него самого, например: при попытке деления на ноль или при обращении к несуществующему адресу. Обработчик исключительных состояний работает в контексте процесса и может обращаться к адресному пространству процесса, а также блокировать процесс, если это необходимо.

Ловушки или ***программные прерывания*** происходят во время выполнения процессом особых инструкций, например в процессе перехода в системные вызовы, и обрабатываются синхронно в контексте процесса.

Системные вызовы

Программный интерфейс ОС определяется набором ***системных вызовов***, предоставляемых ядром пользовательским процессам. Стандартная библиотека C, подключаемая по умолчанию ко всем программам пользователя, содержит ***процедуру встраивания*** для каждого системного вызова. Когда программа делает системный вызов, вызывается соответствующая ему процедура встраивания. Она передает номер системного вызова (идентифицирующего каждый вызов ядра) в пользовательский стек и затем вызывает специальную инструкцию ***системного прерывания***. Функция этой инструкции заключается в изменении режима выполнения на режим ядра и передачи управления обработчику системного вызова.

Системные вызовы выполняются в режиме ядра, но в контексте процесса. Следовательно, они имеют возможность доступа к адресному пространству процесса. С другой стороны, они могут обращаться и к стеку ядра этого процесса.

Прерывания

Основная функция прерываний заключается в том, чтобы обеспечить взаимодействие периферийных устройств, с процессором, информируя его о завершении работы задачи, ошибочных состояниях и других событиях, требующих немедленного внимания. Функция, обслуживающая прерывания называется ***обработчиком прерывания*** или ***процедурой обслуживания прерывания***. Обработчик работает в режиме ядра и системном контексте. Т.к. прерванный процесс обычно не имеет никакого отношения к произошедшему в системе прерыванию, обработчик не должен обращаться к контексту процесса.

Что такое процесс?

Что такое процесса в ОС UNIX? ***Процесс*** - это экземпляр выполняемой программы. Процесс - это нечто, выполняющее программу и создающее среду для ее функционирования. Процесс - это основная единица расписания, т.к. только один процесс может в один момент времени занимать процессор в однопроцессорной системе. Кроме этого процесс старается перехватить ресурсы системы, такие как различные устройства или память. Он также запрашивает системные службы, которые выполняются для него и от его имени ядром.

Каждый процесс имеет определенное время жизни. Большинство процессов создаются при помощи системного вызова **`fork`** и работают до вызова **`exit`.** Во время функционирования процесс может запускать одну или несколько программ, используя системный вызов **`exec`**.

В UNIX процессы иерархически строго упорядочены. Каждый процесс имеет одного ***родителя*** (***parent***) и может иметь также одного или нескольких ***потомков***(***child***). Процесс **`init`** (названный так, потому что он запускает программу /etc/init) является первым прикладным процессом, создаваемым во время загрузки системы. Этот процесс порождает все остальные прикладные процессы. Если какой-либо процесс завершен и после него остаются функционирующие процессы-потомки, то они становятся сиротами и наследуются процессом **`init`**.

Процесс состоит из:

- Инструкций (сегмент кода);
- Данных (сегмент данных);
- Адресного пространства;
- Информации о процессе (элемент таблицы процессов ядра).

Адресное пространство

Выполнение процесса заключается в точном следовании набору инструкций. Процесс выполняет последовательность инструкций в ***адресном пространстве***. Адресное пространство процесса представляет собой набор адресов памяти, к которым он имеет доступ. Процесс отслеживает последовательность выполняемых инструкций при помощи счетчика команд (это аппаратный регистр). Более версии UNIX поддерживают несколько счетчиков команд (***нитей***), т. е. могут существовать несколько параллельно выполняемых последовательностей инструкций в одном процессе.

Адресное пространство процесса является виртуальным, и обычно только его часть соответствует участкам в физической памяти. Ядро хранит содержимое адресного пространства процесса в ***областях свопинга*** (***swap areas***), находящихся обычно на локальных дисках. Подсистема управления памятью ядра переключает ***страницы памяти*** (блоки фиксированного размера) процесса между этими областями по мере необходимости.

Таблица процессов ядра

В различных реализациях UNIX ядро имеет массив фиксированного размера, называемый ***таблицей процессов***. Размер этого массива зависит от максимального количества процессов, которые одновременно могут быть запущены в системе.

Каждому процессу в таблице процессов ядра соответствует свой элемент. Таблицу процессов можно рассматривать как массив структур. Вообще то информация о процессе содержится в двух структурах данных:

- области ***u***;
- структуре ***proc***.

Область ***u*** содержит данные необходимые только в период выполнения процесса. Структура ***proc*** содержит информацию, которая может потребоваться даже в том случае, если процесс не выполняется.

Рассмотрим основные поля области ***u***:

- ***блок управления процессом*** используется для хранения аппаратного контекста, когда процесс не выполняется;
- указатель на структуру ***proc***;
- ***Идентификатор пользователя*** или ***реальный идентификатор пользователя*** (***UID*** или ***RID***) Идентификатор пользователя (реальный идентификатор), запустившего данный процесс. Каждый пользователь системы имеет свой уникальный номер, называемый ***идентификатором пользователя*** (***UID***).
- ***Эффективный идентификатор пользователя*** (***EUID***). Эффективный идентификатор служит для определения прав доступа процесса к системным ресурсам (ресурсам файловой системы). Обычно ***UID*** равен ***EUID***, т.е. процесс имеет в системе те же права, что и пользователь, запустивший его. Однако можно дать процессу больше прав, установив флаг ***SUID***. В этом случае эффективному

идентификатору присваивается значение идентификатора владельца исполняемого файла (если владелец файла *root*, то процесс получит права ***root***);

- ***Идентификатор группы*** или ***реальный идентификатор группы*** (***GID*** или ***RGID***) Системный администратор создает различные группы пользователей, также имеющие уникальный ***идентификатор группы*** (***GID***). Идентификатор группы равен идентификатору первичной или текущей группы пользователя, запустившего процесс. Эффективный идентификатор служит для определения прав доступа к системным ресурсам.

- ***Эффективный идентификатор группы*** (***EGID***). Так же как и для эффективного идентификатора пользователя, его можно установить равным идентификатору группы владельца исполняемого файла (при установки флага ***GUID***);

- **Входные аргументы** и ***возвращаемые значения*** (или коды ошибок) текущего системного вызова;

- ***Обработчики сигналов***;

- и т.д.

Рассмотрим основные поля структуры ***proc***:

- ***Идентификатор процесса*** (***PID***) Каждый процесс имеет уникальный идентификатор, позволяющий ядру системы различать процессы. Когда создается новый процесс, ядро присваивает ему следующий свободный идентификатор (не ассоциированный ни с каким процессом);

- ***Идентификатор родительского процесса*** (***PPID***) Идентификатор процесса, породившего данный процесс;

- ***Текущее состояние процесса***;

- ***Информация определяющая положение процесса в очереди планировщика*** (***или в очереди ожидания***);

- ***Приоритет процесса*** (***Nice Number***) Относительный приоритет процесса, который учитывается ***планировщиком***. Относительный приоритет не изменяется системой на протяжении всей жизни процесса. Фактически же распределение процессора определяется ***приоритетом выполнения***, который зависит от нескольких факторов, в частности от заданного относительного приоритета, и который динамически обновляется ядром;

- ***Информация об обработке сигналов: маски игнорируемых, блокируемых и обрабатываемых сигналов***;
- ***Информация по управлению памятью***;
- и ***т.д.***

Аппаратный контекст

Каждый процесс также имеет набор регистров, которые соответствуют реальным аппаратным регистрам. Ядро хранит регистры процесса, выполняющегося в текущий момент времени в аппаратных регистрах, и сохраняет регистры остальных процессов в ***контекстах*** процессов (специальных структурах данных, отводимых для каждого процесса).

Аппаратный контекст включает содержимое регистров общего назначения, а также набора специальных системных регистров.

- ***Программный счетчик*** (***program counter***, ***PC***). Хранит адрес следующей выполняемой инструкции.
- ***Указатель стека*** (***stack pointer***, ***SP***). Содержит адрес верхнего элемента стека.
- ***Слово состояния процессора*** (***processor status word***, ***PSW***). Содержит несколько битов с информацией о состоянии системы, в том числе о текущем и предыдущем режимах выполнения, текущем и предыдущем уровнях приоритетов прерываний, а также биты переполнения и переноса.
- ***Регистры управления памятью***, в которых хранится адреса таблиц трансляции адресов процесса.
- ***Регистры сопроцессора*** (***Floating point unit***, ***FPU***).

Машинные регистры содержат аппаратный контекст текущего выполняемого процесса. Когда происходит переключение контекста эти регистры в ***блоке управления процессом*** (***process control block***, ***PCB***). Ядро выбирает следующий процесс для выполнения и загружает его аппаратный контекст из блока PCB.

Состояния процесса

В системе UNIX процессы могут находиться определенное *состояние*. Переход из одного состояния в другое происходит вследствие различных событий. На рисунке (*Рисунок 2.3*) показаны важнейшие состояния процесса в системе UNIX, а также события, являющиеся причинами изменения состояния.

Рассмотрим состояния процесса:

- ***Процесс только что создан вызовом*** **`fork`**;
- ***Процесс не выполняется, но готов к выполнению*** (***ready***). Процесс находится в очереди на выполнение и обладает всеми необходимыми ему ресурсами, кроме процессора;
- ***Процесс находится в состоянии сна*** (***asleep***). Ожидает недоступного в данный момент ресурса. Не претендует на процессор;
- ***Процесс выполняется в режиме задачи*** (***running***) При этом процессором выполняются прикладные инструкции данного процесса;
- ***Процесс выполняется в режиме ядра*** (***running***) При этом процессором выполняются системные инструкции ядра ОС от имени процесса;
- ***Процесс перешел в состояние зомби***. Как такового процесса не существует, но в таблице процессов остается запись, содержащая код возврата доступный для родительского процесса.

Системный вызов **`fork`** создает процесс, который сначала находится в состоянии *инициализации*. После того как инициализация процесса завершится, он переходит в состояние ***готовности к выполнению*** (***ready***). В этом состоянии он ожидает своей очереди на обслуживание.
В какой-то момент времени ядро:

- выбирает этот процесс на ***выполнение*** (***running***);
- загружает его аппаратный контекст в системные регистры;
- и передает ему управление.

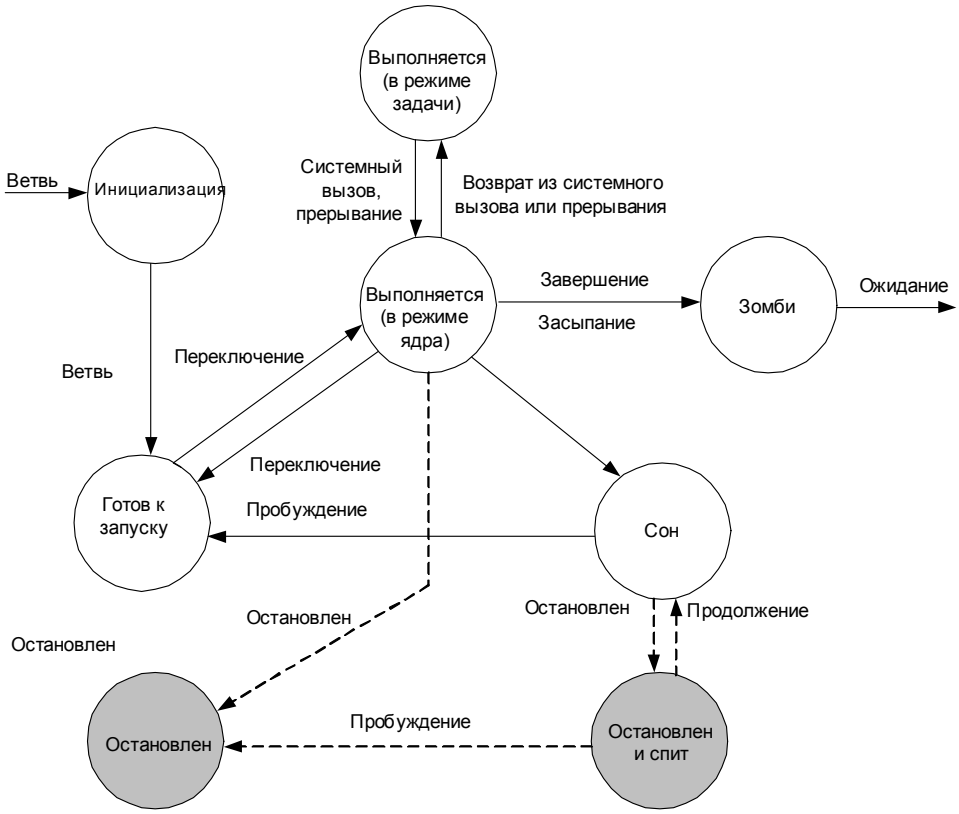
Таким образом, процесс начинает выполняться в режиме ядра. Следующее изменение состояния зависит от того, чем занимался процесс до того как достиг своей очереди на выполнение. Если он только что создан или выполняет код программы, то он переводится в режим задачи. Если он был блокирован в ожидании ресурса во время выполнения системного вызова, то он продолжит выполнение в режиме ядра.

Процесс, выполняющийся в режиме задачи, может перейти в режим ядра в результате системного вызова или прерывания, а возвращается обратно после завершения соответствующей процедуры обработки. Процессу бывает необходимо ждать некоторого события или освобождения ресурса недоступного в данный момент. Такое ожидание реализуется вызовом **`sleep`.** Этот вызов переводит процесс в

ожидающее (спящее) состояние и помещает его в очередь спящих процессов. После того как событие произойдет или ресурс станет, доступен, ядро разбудит процесс и снова сделает его готовым к выполнению.

Завершение работы процесса происходит в результате системного вызова exit или по сигналу. По окончании работы процесса ядро освобождает все ресурсы завершаемого процесса (сохраняя информацию о статусе выхода и использовании ресурсов) и оставляет процесс в ***состоянии зомби*** (***zombie***). Процесс остается в этом состоянии до тех пор, пока его процесс-родитель не вызовет **`wait`**, который уничтожит процесс и передаст статус выхода родительскому процессу.

Рисунок 2.3 Диаграмма состояний процесса



Планирование

UNIX & LINUX является системой разделения времени, это означает, что каждому процессу вычислительные ресурсы выделяются на ограниченный промежуток времени, после чего они предоставляются другому процессу и т.д. Максимальный временной интервал, на

который процесс может захватить процессор, называется ***квантом времени*** (***time quantum time*** или ***slice***). Таким образом, создается иллюзия, что процессы выполняются одновременно, хотя в действительности в каждый момент времени выполняется только один процесс (на однопроцессорной машине).

Центральный процессор является общим ресурсом, разделяемым между всеми процессами системы. Планировщик является компонентом ОС, определяющим, какой из процессов должен выполняться в данный момент времени и как долго он может занимать процессор.

При рассмотрении планировщика UNIX необходимо обсудить:

- правила, используемые при принятии решения о том, какой из процессов следует назначить на выполнение и когда произвести переключение на выполнение другого процесса;
- реализацию, представляющую собой набор структур данных и алгоритмов, используемых для проведения в жизнь этих правил.

На низком уровне планировщик переключает процессор от одного процесса к другому. Это действие называется переключением контекста. Ядро системы сохраняет аппаратный контекст текущего выполняющегося процесса в ***блоке управления процессом*** (***process control block***, ***PCB***). Контекст - это "моментальный снимок" текущих значений регистров общего назначения, регистров управления памятью и других специальных регистров процесса (см. *“Аппаратный контекст”*). После завершения процедуры сохранения ядро системы загружает контекст следующего процесса, готовящегося к выполнению (Контекст загружается из его блока PCB). Это действие приводит к тому, что CPU начинает выполнение нового процесса.

Задачи планировщика

Планировщик должен наиболее оптимально распределять процессорное время между всеми процессами системы. А также, учитывая загруженность системы, обеспечивать приемлемую производительность каждому приложению при общей загруженности системы в рамках нормы.

Приложения можно условно разбить на несколько классов, в зависимости от их требований к планированию и к производительности работы:

- ***Интерактивные приложения.*** Приложения типа командных интерпретаторов, редакторов и программ с графическим пользовательским интерфейсом, постоянно взаимодействующих с пользователем. Большую часть времени такие приложения находятся в ожидании действий пользователя, таких как ввод с клавиатуры или манипуляции мышью. После получения ввода приложение должно быстро его обработать, иначе пользователь будет скучать;

- ***Пакетные приложения.*** К ним относятся такие, как сборка программ или вычисления, т. е. программы, не требующие взаимодействия с пользователем и часто выполняющиеся в фоновом режиме. Для таких задач эффективность планирования определяется временем завершения их работы при функционировании других процессов, сравниваемое со временем их выполнения если бы они были единственной задачей в системе;

- ***Приложения реального времени.*** Это - класс задач, для которых время является критическим. Обычно все они требуют гарантированного времени реакции.

На рабочей станции одновременно могут выполняться сразу несколько типов приложений. А планировщик должен оптимально удовлетворить требования каждого из них.

Приоритеты процессов

Механизм планирования в UNIX основан на использовании ***приоритетов***, назначенных процессам. Каждый процесс имеет ***приоритет планирования***, который может изменяться во время выполнения. Процесс, обладающий наивысшим приоритетом, планировщик назначает на выполнение. Используется ***вытесняющее планирование.*** Планировщик поставит процесс, имеющий приоритет выше приоритета выполняемого процесса, на выполнение, даже если у выполняемого процесса не израсходован его ***квант времени***.

Традиционное само ядро UNIX является не вытесняющим. Если процесс выполняется в режиме ядра (например, в течение исполнения системного вызова или прерывания), то ядро не заставит такой процесс уступить процессор высокоприоритетному процессу. Выполняющийся процесс может только добровольно освободить процессор в случае своего блокирования, ожидая необходимый ресурс. Реализация не вытесняющего ядра решает проблемы синхронизации, связанных с доступом нескольких процессов к одним и тем же структурам данных ядра.

Приоритет процесса задается любым целым числом, лежащим в диапазоне от 0 до 127. Чем меньше число, тем выше приоритет процесса. Приоритеты от 0 до 49 зарезервированы для ядра, следовательно, прикладные процессы могут обладать приоритетами от 50до 127.

Приоритет в режиме задачи зависит от двух факторов:

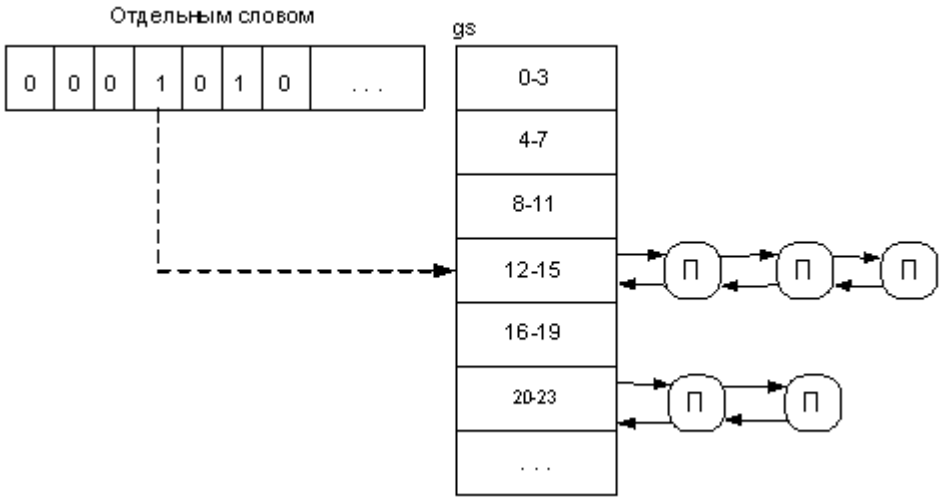
- ***любезности*** (***nice***);
- и ***величины использования процессора***.

Степень любезности (***nice value***) является числом в диапазоне от 0 до 39 (со значением 20 по умолчанию). Увеличение значения приводит к уменьшению приоритета. Фоновым процессам автоматически задаются более высокие значения степени любезности. Уменьшить эту величину, для какого либо процесса, может только root, поскольку при этом его приоритет становится выше.

Реализация планировщика

Планировщик содержит массив, состоящий из 32-х очередей выполнения (см. *Рисунок 2.4*). Каждая очередь соответствует четырем приоритетам. Таким образом, очередь 0 используется для приоритетов 0-3, очередь 1 для приоритетов 4-7 и т.д.

Рисунок 2.4 Структуры данных планировщика



Процесс, обладающий наивысшим приоритетом, запускается всегда, если только текущий процесс не выполняется в режиме ядра. Каждому процессу назначается квант времени фиксированного размера. Через каждые 100 ms ядро (через отложенный вызов) вызывает процедуру для постановки на выполнение очередного процесса из одной и той же очереди. Если в состоянии готовности к выполнению окажутся

процессы с более высоким приоритетом, то они будут назначены на выполнение. Если все остальные процессы готовые к выполнению, находятся в очередях с более низким приоритетом, то текущий процесс продолжит выполнение даже после вырабатывания отведенного ему кванта времени.

Существует три ситуации, при которых возникает переключение контекста:

- Если текущий процесс блокируется в ожидании ресурса или завершает работу. При этом происходит свободное переключение контекста;
- Если в результате, полученном процедурой пересчета приоритетов, оказалось, что другой процесс обладает более высоким приоритетом по сравнению с текущим;
- Если текущий процесс или обработчик прерываний разбудил более приоритетный процесс.

Управление процессами

В UNIX нет системного вызова, который создавал бы новый процесс для выполнения конкретной программы. Соответственно система предоставляет два различных системных вызова: один для создания процесса, а другой для запуска новой программы.

Новый процесс порождается с помощью системного вызова **`fork`**.

Порожденный, или дочерний процесс является почти точной копией процесса, выполнившего этот вызов, или родительского процесса. Дочерний процесс наследует следующие атрибуты родительского процесса:

- Идентификатора пользователя и группы;
- Переменные окружения;
- Диспозицию сигналов и их обработчики;
- Текущий каталог;
- Маску создания файлов;
- Управляющий терминал;
- Все файловые дескрипторы.

Кроме того, виртуальная память дочернего процесса не отличается от образа родительского процесса: такие же сегменты кода, данных, стек. После возврата из **`fork`**, который происходит и в родительский и в дочерний процесс, оба начинают выполнять одну и ту же инструкцию.

Теперь перечислим отличия между ними:

- Дочернему процессу присваивается уникальный идентификатор ***PID***;
- Идентификатор родительского процесса (***PPID***) у порожденного процесса равен ***PID*** родителя;
- Значение, возвращаемое системным вызовом **`fork()`** различно для родителя и потомка;
- Дочерний процесс свободен от сигналов, ожидающих доставки.

Возврат из функции **`fork`** происходит как в родительский, так и в дочерний процесс. При этом возвращаемое родителю значение равно PID дочернего процесса, а дочерний, в свою очередь, получает значение 0. Если **`fork`** возвращает -1, то произошла ошибка (в этом случае возврат происходит только в процесс, выполнивший системный вызов).

После порождения нового процесса родитель и потомок параллельно выполняют различные функции.

Теперь рассмотрим системный вызов для загрузки другого исполняемого файла **`exec`**.

При этом наследуются следующие атрибуты:

- Идентификаторы процесса ***PID*** и ***PPID***;
- Идентификаторы пользователя и группы;
- Эффективные идентификаторы пользователя и группы (если флаг ***SUID*** и ***SGID*** не установлен);
- Текущий каталог;
- Маску создания файлов;
- Управляющий терминал;
- Файловые дескрипторы.

При порождении процесса потомка, родителю бывает интересно знать о завершении выполнения потомка. Процессам бывает необходимо синхронизовать свое выполнение с выполнением других процессов.

Одним из способов такой синхронизации является обработка родителем сигнала ***SIGCHLD***, который посылается ему при окончании выполнения потомка. Вызов wait позволяет заблокировать выполнение процесса, пока кто-либо из его потомков не прекратит существование.

Процесс завершает свое выполнение с помощью функции **`exit`**. Надо либо использовать системный вызов **`exit`**, а если завершение процесса вызвано получением сигнала, то функцию **`exit`** вызовет само ядро. Функция выполняет следующие действия:

- Отключает все сигналы;
- Закрывает все открытые файлы;
- Изменяет состояние процесса на зомби
- Освобождает адресное пространство;
- Посылает родителю сигнал ***SIGCHLD***.

После завершения **`exit`** процесс находится в состоянии зомби. Этот вызов не освобождает структуру proc завершенного процесса, так как родителю может потребоваться информация о статусе выхода и использовании ресурсов. За освобождение структуры ***proc*** потомка отвечает его родитель. Проблема возникнет, если процесс завершится раньше своего родителя, а он не вызовет wait.

Нити

Использования процессов обладает двумя важными ограничениями:

- во-первых, некоторым приложениям необходимо выполнять несколько независимых друг от друга задач, при этом используя одно и тоже адресное пространство, и другие ресурсы;
- во-вторых, невозможно реализовать преимущества многопроцессорных систем, так как процесс способен использовать только один процессор одновременно.

Но в тоже время существуют методики, позволяющие процессам совместно использовать адресное пространство и др. ресурсы. В разных реализациях UNIX используется различная терминология:

- нити ядра;
- прикладные нити;
- прикладные нити, поддерживаемые ядром;

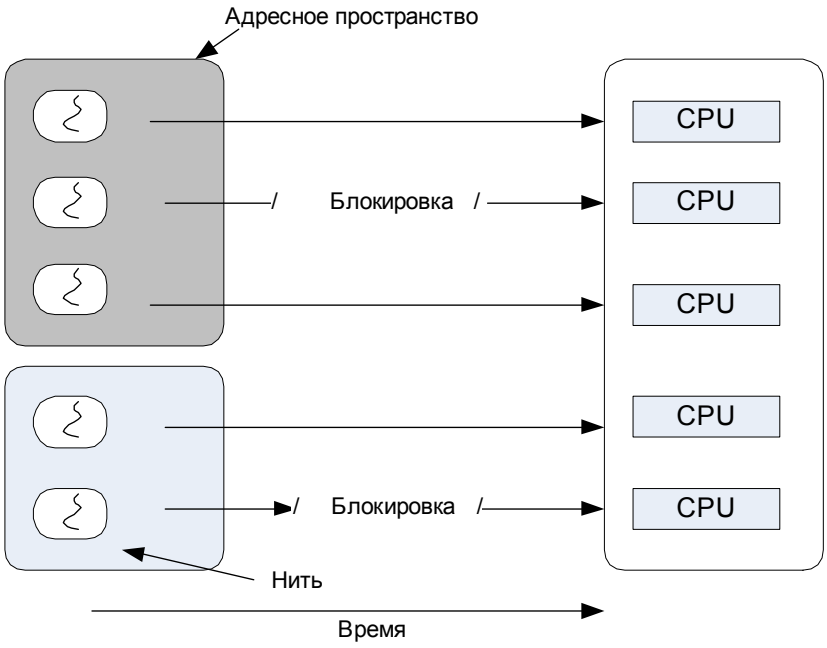
- C-threads;
- pthreads и "легковесные" процессы (lightweight process).

Причины появления нитей

Многие программы выполняют отдельные независимые задач. Например, сервер баз данных находится в режиме приема и обработки множества запросов. Но если поступающие запросы не требуется обрабатывать в определенном порядке, то их можно считать независимыми задачами, функционирующими параллельно. В традиционных системах UNIX такие программы используют несколько процессов, что приводит к возникновению следующих проблем:

- при создании процессов происходят перегрузки системы т.к. вызов **`fork`** является затратным. Память копируется от родительского процесса к дочернему, копируются все дескрипторы и т.д. Существующие реализации используют метод копирования при записи (copy-on-write). Хотя и применяется оптимизация - адресное пространство родительского процесса не копируется, пока оно не понадобится клиенту-ребенку, этот вызов остается ресурсоемким;
- каждый процесс находится в своем адресном пространстве, поэтому для взаимодействия между ними необходимо использовать специальные средства.

Рисунок 2.5 Многонитевые процессы в многопроцессорной системе



Определим понятие независимого вычислительного блока в качестве части общей задачи приложения. Такие блоки мало взаимодействуют друг с другом, поэтому не требуют больших затрат по синхронизации. Приложение может содержать один или несколько таких блоков. Такой вычислительный блок называется нитью.

Нити (***потоки***) помогают решить обе проблемы. Их называют ***облегченными процессами*** (***lightweight processes***), поскольку они проще чем процесс. Создание нити может занимать по времени меньше одной десятой создания процесса.

Все нити одного процесса совместно используют его глобальные переменные, поэтому им легко обмениваться информацией, но это приводит к необходимости синхронизации. Общими становятся не только глобальные переменные. Все нити одного процесса разделяют:

- Инструкции процесса;
- Большую часть данных;
- Открытые файлы (дескрипторы);
- Обработчики сигналов и вообще настройки для работы с сигналами;
- Текущий рабочий каталог;
- Идентификаторы пользователя и группы.
- Каждый поток имеет свой собственный:
- Идентификатор потока;
- Набор регистров, включая PC и указатель стека;
- Стек (для локальных переменных и адресов возврата);
- Errno
- Маску сигналов;
- Приоритет;
- и т.д.

Одновременность и параллельность

Введем термины ***одновременность*** (***concurrency***) и ***параллельность*** (***parallelism***). Параллельность приложения это достигнутая им степень параллельного выполнения, которая ограничена количеством процессоров, доступных приложению (см. *Рисунок 2.5*). А термин

одновременность описывает максимальную степень параллельности, которую теоретически может достичь приложение при неограниченным количеством доступных процессоров.

Ядро системы обеспечивает ***системную одновременность*** при помощи распознавания нескольких нитей внутри процесса (***горячие нити***, ***hot threads***) и планирования их выполнения независимо. Ядро разделяет такие нити между доступными процессорами.

Приложения могут обеспечивать прикладную одновременность за счет использование нитевых библиотек прикладного уровня. Такие нити (***холодные нити***, ***cold threads***) не распознаются ядром системы и должны управляться и планироваться самими приложениями.

Во многих системах реализована модель ***двойной одновременности***, которая совмещает в себе ***системную*** и ***прикладную одновременность***. В этой модели нити процесса делятся на те, которые ядро распознает, и те, которые реализованы в библиотеках прикладного уровня и ядру не видимы.

Типы нитей

Нить - это динамический объект, в процессе представленный одной точкой управления и выполняющий последовательность команд, отсчитываемую от этой точки (выполняемая параллельно в рамках процесса часть программы). Ресурсы, включающие адресное пространство, открытые файлы, полномочия, квоты и т.д., используются всеми нитями процесса совместно. Каждая нить, кроме того, обладает собственными объектами, такими как указатель команд, стек или контекст ресурсов. В традиционных системах UNIX процесс имеет единственную выполняющуюся нить. Многонитевые системы позволяют каждому процессу иметь более одной выполняющейся нити.

Опишем различные типы нитей.

Нити ядра

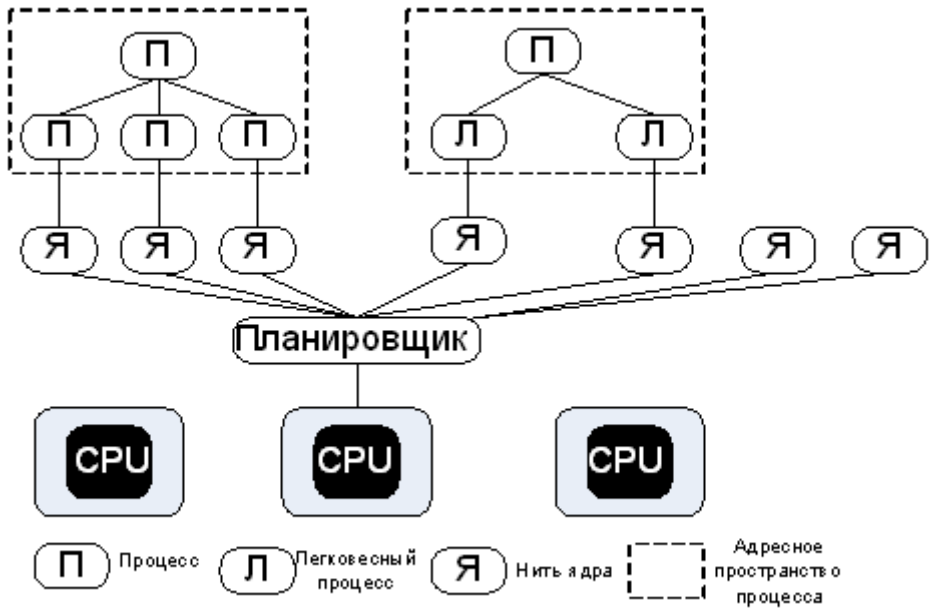
Нити ядра не требуют связи с каким-либо прикладным процессом. Они создаются и уничтожаются ядром и внутри ядра по мере необходимости и отвечают за выполнение определенных функций. Такие нити используют совместно доступные области кода глобальные данные ядра, но обладают собственным стеком в ядре. Они могут независимо назначаться на выполнение и используют стандартные механизмы синхронизации ядра, такие как **`sleep`** или **`wakeup`**. Нити ядра применяются для выполнения таких операций, как асинхронный ввод-вывод, они также являются малозатратными при создании и

использовании. Переключение контекста между нитями ядра происходит быстро, что исключает необходимость обновлять отображение памяти.

Легковесные процессы

Легковесный процесс (или ***LWP***, ***lightweight process***) - это прикладная нить, поддерживаемая ядром. ***LWP*** - это абстракция высокого уровня, основанная на понятии нити ядра. Каждый процесс может иметь один или более ***LWP***, любой из которых, поддерживается отдельной нитью ядра (*Рисунок 2.6*). Легковесные процессы планируются на выполнение независимо от процесса, но совместно разделяют адресное пространство и другие ресурсы процесса. Многонитевые процессы применяются в тех случаях, когда каждая нить является полностью независимой и редко взаимодействует с другими нитями. Если доступ к каким-либо данным производится одновременно несколькими ***LWP***, необходимо обеспечить синхронизацию доступа. Для этого ядро системы предоставляет средства блокировки. Такими средствами являются взаимные исключения семафоры и условные переменные.

Рисунок 2.6 Легковесные процессы



Многие операции над легковесными процессами, например создание, уничтожение и синхронизация, требует применения системных вызовов. Но системные вызовы являются затратными операциями. Так как каждый вызов требует двух ***переключений режима***:

- сначала из режима задачи в режим ядра;
- и обратное переключение после завершения работы функции.

Когда легковесным процессам необходимо часто пользоваться разделяемыми данными, то затраты на синхронизацию могут свести на нет увеличение производительности от их применения.

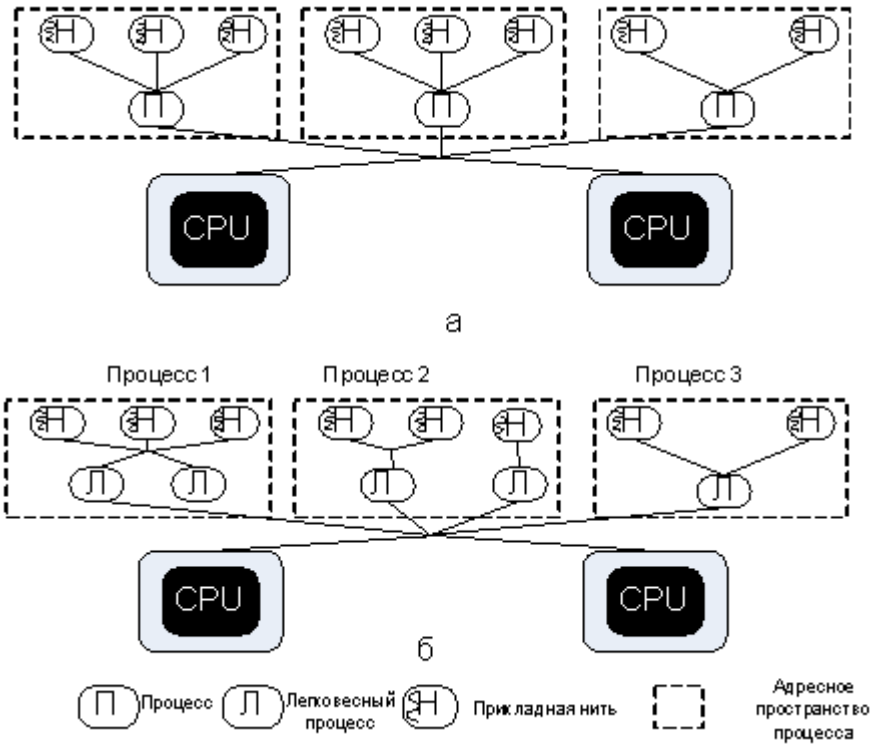
Таким образом, что хотя ядро системы предоставляет механизмы создания, синхронизации и обработки легковесных процессов, но за их правильное применение отвечает программист.

Прикладные нити

Существует возможности поддержки нитей полностью на прикладном уровне (см. *Рисунок 2.7*), при этом ядру об их существовании ничего не известно.

Библиотеки ***C-threads*** системы ***Mach*** и ***pthreds*** стандарта ***POSIX*** содержат все необходимые функции для создания, синхронизации, планирования и обработки нитей без какой-либо специальной помощи ядра. Вследствие этого функционирование таких нитей является необычайно быстрым.

Рисунок 2.7 Применение пользовательских нитей: а - прикладные нити обычных процессов; б - мультиплексирование прикладных нитей



Прикладной контекст нити может сохраняться и восстанавливаться без вмешательства ядра. Каждая прикладная нить обладает собственным стеком в адресном пространстве процесса, областью для хранения контекста регистров прикладного уровня и другой важной информации, такой как маски сигналов. Библиотека планирует выполнение и переключает контекст между прикладными нитями, сохраняя стек и состояние регистров следующей по расписанию нити. Прикладные нити не являются по настоящему планируемыми задачами, так как ядро ничего о них не знает.

Ядро просто планирует выполнение процесса (или ***LWP***). Если процесс или ***LWP*** вытеснен кем-то, такой же участи и все его нити. Если процесс обладает всего одним ***LWP*** (или если прикладные нити реализованы на однонитевой системе), будут заблокированы все его нити.

Нитевая библиотека также включает в себя объекты синхронизации, обеспечивающие защиту совместно используемых структур данных.

ЛАБОРАТОРНАЯ РАБОТА 2

ПРОЦЕССЫ И НИТИ (ПРОГРАММИРОВАНИЕ)

Теория

Создание процесса

Новый процесс порождается с помощью системного вызова **fork**:

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void );

main()
{
   int pid;
   pid=fork();
   if( pid == -1 )  {    printf ("Произошла ошибка\n");  exit(1); }
   if( pid == 0 )    {  printf ("Выполняется потомок\n");            }
   else                   { printf("Выполняется родитель\n");             }
}
```

Теперь рассмотрим системный вызов для загрузки другого исполняемого файла, **exec**:

```
#include <unistd.h>

int exec(const char *path, const char *arg0, ….., const char *argN);
```

Вызов **wait** позволяет заблокировать выполнение процесса, пока кто-либо из его потомков не прекратит существование.

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *stat_loc);
```

Создание потока

Функция pthread_create

При запуске программы вызовом **exec** создается единственный поток, называемый ***начальным*** или ***главным*** (***initial thread***) потоком. Дополнительные потоки создаются вызовом ***pthread_creat***e:

```
#include <pthread.h>

int pthread_create(pthread_t *tid, const pthread_attr_t, void *(*func)
(void),
void *arg);
/* Возвращает 0 в случае успешного завершения и положительное значение Exxx
- в случае ошибки */
```

Каждый поток процесса обладает собственным идентификатором потока, который имеет тип **pthread_t**. При успешном создании нового потока его идентификатор возвращается через указатель **tid**.

Каждый поток обладает некоторым количеством атрибутов:

- приоритетом;
- начальным размером стека;
- признаком демона;
- и т.д.

При создании потока эти атрибуты могут быть указаны с помощью переменной типа **pthread_attr_t**, значение которой имеет более высокий приоритет, чем значение по умолчанию. Обычно используют значение по умолчанию. При этом аргумент **attr** является нулевым указателем.

Наконец, при создании потока мы должны указать функцию, которую он будет выполнять, - ***начальную функцию потока*** (***thread start function***). Поток запускается вызовом этой функции и завершается либо явно (вызовом **pthread_exit**), либо неявно (возвратом из этой функции). Адрес функции указывается в аргументе func, и вызывается она с единственным аргументом - указателем **arg**. Если функции нужно передать несколько аргументов, следует упаковать их в структуру и передать ее адрес в качестве единственного аргумента начальной функции.

Обратите внимание на объявление **func** и **arg**. Функция принимает один аргумент - указатель типа **void**, и возвращает один аргумент - такой же указатель. Это дает нам возможность передать потоку указатель, на что угодно и получить в ответ такой же указатель.

Функции ***POSIX*** для работы с потоками обычно возвращают 0 в случае успешного завершения работы и ненулевое значение в случае ошибки. В отличие от большинства системных функций, возвращающих -1 в случае ошибки и устанавливающих значение **errno** равным коду ошибки, функции **pthread** возвращают положительный код ошибки. Например, если **pthread_create** не сможет создать новый поток из-за превышения системного ограничения на потоки, эта функция вернет

значение **EAGAIN**. Функция **pthread** не устанавливает значение переменной **errno**. Несоответствия при их вызове не возникнет, поскольку ни один из кодов ошибок не имеет 0 значение (**<sys/errno.h>**).

Функция pthread_join

Мы можем ждать завершения какого-либо потока, вызвав **pthread_join**. Сравнивая потоки с процессам, можно сказать, что **pthread_create** аналогична **fork**, а **pthred_join** - **waitpid**:

```
#include <pthread.h>

int pthread_join(pthread_t tid, void **status);
/* Возвращает 0 в случае успешного завершения и положительное значение Exxx
- в случае ошибки */
```

Мы должны указать идентификатор потока, завершения которого ожидаем. К сожалению, невозможно задать режим ожидания завершения нескольких потоков (аналога ***waitpid*** с идентификатором процесса - 1 нет). Если указатель status не нулевой, возвращаемое потоком значение (указатель на объект) сохраняется в ячейке памяти, на которую указывает status.

Функция pthread_self

У каждого потока имеется свой идентификатор, уникальный в пределах данного процесса. Идентификатор возвращается **pthread_create** и используется при вызове **pthread_join**. Поток может узнать свой собственный идентификатор вызвав **pthread_self**:

```
#include <pthread.h>

pthread_t pthread_self(void);
/* возвращает идентификатор вызвавшего потока */
```

Вызов **pthread_self** является аналогом **getpid** для процессов UNIX.

Функция pthread_detach

Поток может являться как присоединенным (по умолчанию), так и отсоединенным. При завершении присоединенного потока его идентификатор и статус завершения сохраняются до тех пор, пока какой-либо другой поток данного процесса не вызовет **pthread_join**. Отсоединенный поток функционирует аналогично процессу-демону. После его завершения все ресурсы освобождаются. Никакой другой поток не может ожидать его завершения. Если нужно, чтобы один поток ждал завершения другого, то лучше оставить последний присоединенным.

Функция **pthread_detach** делает данный поток отсоединенным:

```
#include <pthread.h>

int pthread_detach(pthread_t tid);
/* Возвращает 0 в случае успешного завершения и положительное значение Exxx
- в случае ошибки */
```

Эта функция вызывается потоком при необходимости изменить собственный статус в форме

```
pthread_detach(pthread_self());
```

Функция pthread_exit

Одним из способов завершения потока является вызов **pthread_exit**:

```
#include <pthread.h>

void pthread_exit(void *status);
/*ничего не возвращает вызвавшему потоку*/
```

Если поток не является отсоединенным, его идентификатор и статус завершения сохраняются для возвращения другому потоку, который может вызвать **pthread_join**.

Указатель status не должен быть установлен на локальный объект вызвавшего потока (типа автоматической переменной), поскольку этот объект уничтожается при завершении потока.

Поток может быть завершен двумя другими способами:

начальная функция потока (третий аргумент **pthread_create**) может вызвать **return**. Поскольку эта функция должна объявляться как возвращающая указатель на тип **void**, это возвращаемое значение становится статусом завершения потока;

функция **main** процесса может завершить работу или один из потоков может вызвать **exit** или **_exit**. При этом процесс завершает работу немедленно, вместе со всеми своими потоками.

Чему нужно научиться ?

*Нужно научиться создавать процессы и нити (потоки). Для этого используйте системные вызовы **fork()**, **exec()**, **wait()**, **exit()**, **sleep()**.*

Задания

Уровень 1 (A)

Написать программы родителя и ребенка (father.c и son.c).

(Чтобы сохранить состояние таблицы процессов в файле использовать системный вызов **`system ("ps -abcde > file")`**).

Компиляция: **`gcc -o name.out name.c`**.

Уровень 2, 3 (A)

Написать программу взаимодействия процессов родственников (father и son):

- процесс-родитель (father) порождает процесс (son) и ожидает его завершения;
- процесс-родитель (father) порождает процесс (son) и, не ожидая его завершения, завершает свое выполнение. При этом посмотрите, как изменится ***PPID*** процесса son;
- процесс-родитель (father) порождает процесс son и не ожидает его завершения, а процесс son завершает свое выполнение. При этом покажите, что появился процесс зомби.

Зафиксируйте результаты работы этих трех случаев в файле (перенаправьте вывод с терминала в файл).

Уровень 1, 2, 3 (B)

ВНИМАНИЕ!!! *Для компиляции вашего файла используйте*
`gcc -o OUTPUT /lib/libthread.so.0 INPUT.C`

Написать программу, формирующую несколько нитей.

Пусть каждая нить в бесконечном цикле:

- выводит на экран свое имя;
- увеличивает на 1 значение локальной переменной;
- засыпает (**`sleep(5)`**).

Скелет кода для задания A

father.c

```
#include <stdio.h>
main()
{
   int pid, ppid, status;

   pid=getpid();
   ppid=getppid();
   printf("\n\n FATHER PARAM: pid=%i  ppid=%i \n", pid,ppid);

   if(fork()==0) execl("son","son", NULL);
   system("ps xf");
   wait(&status);
}
```

son.c

```
#include <stdio.h>

main(){
int pid,ppid;

pid=getpid();
ppid=getppid();

printf("\n SON PARAM:  pid=%i  ppid=%i \n",pid,ppid);
sleep(15);
}
```

Скелет кода для задания B

```
#include <pthread.h>

void* thread1 (void* args){       }
void* thread2 (void* args){       }

pthread_t t1, t2;
```

Чтобы создать и запустить нить используйте функции:

```
  pthread_create (&tn, NULL, threadn, NULL);
  pthread_join (tn, NULL
```

где n - номер нити.

Глава

3

Взаимодействие между процессами (Interprocess Communication, IPC)

Введение

В UNIX каждый процесс выполняется в своем собственном адресном пространстве. Это позволяет исключить "плохое" влияние одного процесса на другой. В сложных программных системах часто используются несколько взаимодействующих процессов, решающих общую задачу.

Процессы могут взаимодействовать:

- **Передавая данные.** Один процесс передает данные другому процессу, объемом от десятков байтов до нескольких мегабайтов.
- **Совместно используя данные.** Можно не копировать информацию, а использовать ее совместно, при этом изменения, сделанные одним процессом, должны быть сразу видны другому. Взаимодействовать могут несколько процессов. При совместном использовании данных процессам нужен протокол взаимодействия, который позволит:
  - сохранить целостность данных;
  - исключить конфликты при доступе данным.
- **Уведомляя о событиях.** Процесс может сообщить другому процессу или группу процессов о наступлении некоторого события. Таким образом, можно синхронизировать выполнения нескольких процессов.

Поэтому процессам надо предоставить возможность общаться и разделять между собой общие ресурсы и данные. Можно решать эту задачу средствами самих процессов, но это неэффективно, а в

многозадачной системы еще и опасно. ОС должна предоставлять ***средства межпроцессного взаимодействия*** (***InterProcess Communication, IPC***).

Средства межпроцессного взаимодействия, присутствующие во всех версиях UNIX, это:

- ***сигналы***,
- ***каналы***,
- ***FIFO*** (***именованные каналы***),
- ***сообщения*** (***очереди сообщений***),
- ***семафоры***,
- ***разделяемая память***.

Последние три типа IPC обычно обобщенно называют ***System V IPC***. Во многих версиях UNIX есть еще одно средство ***IPC - сокеты***, впервые реализованные в BSD UNIX.

Первая внешняя реализации системы UNIX поддерживала следующие средства, которые можно использовать для взаимодействия процессов:

- ***сигналы***
- и ***каналы***.

Сигналы

Общие сведения

Сигналы в системе UNIX используются для того, чтобы:

- ***сообщить процессу о том, что возникло асинхронное событие***;
- или ***необходимо обработать исключительное состояни***е.

Изначально сигналы были разработаны для уведомления об ошибках. В дальнейшем их стали использовать и для IPC, например, для синхронизации процессов или для передачи простейших команд от одного процесса другому.

Например, когда пользователь нажимает комбинацию клавиш Ctrl+C, процессу, с которым пользователь в данный момент интерактивно работает, передается сигнал **`SIGINT`**. Когда процесс завершается, он

отправляет своему процессу-родителю сигнал **SIGCHLD**. В ОС UNIX поддерживается определенное количество сигналов (31 в 4.3BSD и SVR3).

- Процесс может выслать сигнал одному или нескольким процессам, используя системный вызов **kill**.
- Драйвер терминала вырабатывает сигналы в ответ на нажатия клавиш.
- Ядро вырабатывает сигналы для уведомления процесса о возникновении аппаратного исключения или в случаях возникновения определенных ситуаций.

Каждый сигнал имеет определенную по умолчанию реакцию на него, обычно это - завершение процесса. Некоторые сигналы по умолчанию игнорируются, а некоторые приостанавливают процесс. Процесс может установить другую реакцию на сигнал, отличную от реакции заданной по умолчанию. Для этого используется вызов **signal** (System V) или **sigaction** (POSIX.1). В этом случае может происходить следующее:

- запускается обработчик сигнала, определенный разработчиком приложения;
- или сигнал игнорируется;
- или сигнал блокируется.

При блокировке сигнал будет доставлен процессу только после того, как тот будет разблокирован.

Процесс не в состоянии прореагировать на сигнал немедленно. После вырабатывания сигнала ядро системы уведомляет об этом процесс при помощи установки бита в маске ожидающих сигналов, расположенной в структуре **proc** данного процесса. Процесс должен постоянно быть готовым к получению сигнала и ответу на него. Это возможно только в том случае, если он находится в очереди на выполнение. В начале выполнения процесс обрабатывает все ожидающие его сигналы и только затем продолжает работать в режиме задачи.

Стандарты

Программный интерфейс, поведение, а также реализация сигналов отличаются в различных версиях UNIX. ОС предоставляет дополнительные системные вызовы и библиотечные функции для поддержки ранних интерфейсов сигналов, а также для обеспечения обратной (backward) совместимости, и все это только запутывает разработчика.

Оригинальная реализация сигналов в ОС System V была неэффективной и ненадежной. Многие проблемы были решены после появления системы 4.2BSD UNIX, в которой был предложен новый надежный механизм сигналов.

Стандарт POSIX 1003.1 (также известный как POSIX.1) навел некоторый порядок в хаосе различных реализаций сигналов. Он определил стандартный интерфейс, который должны поддерживать все совместимые с POSIX ОС. Однако стандарт не описывает, каким образом этот интерфейс должен быть реализован. Разработчики ОС сами решают, на каком уровне они будут поддерживать рекомендации стандарта:

- в ядре;
- или через прикладные библиотеки;
- или часть там, а часть там.

Генерация и доставка

При работе с сигналами необходимо различать два этапа:

- **Генерация или отправление сигнала.** Сигнал посылается, когда происходит определенное событие, о наступлении которого должен быть уведомлен процесс;
- **Доставка и обработка сигнала.** Сигнал считается доставленным, когда процесс, которому был отправлен сигнал, получает его и выполняет его обработку.

В промежутке между этими моментами сигнал ожидает доставки.

К генерации сигнала могут привести различные ситуации:

- Ядро отправляет процессу (или группе процессов) сигнал при нажатии пользователем определённых клавиш или их комбинаций. Например, нажатие клавиши <Del> (или Ctrl+C) приведёт к отправке сигнала **`SIGINT`**;
- Аппаратные особые ситуации. Эти ситуации определяются аппаратурой компьютера и ядру посылается соответствующее уведомление (например, в виде прерывания). Ядро реагирует на это отправкой соответствующего сигнала процессу, который находился в стадии выполнения, когда произошла особая ситуация;
- Определённые программные состояния системы или её компонентов также могут вызвать отправку сигнала. Эти условия не связаны с аппаратной частью, а имеют чисто программный характер. В

качестве примера можно привести сигнал **SIGALRM**, отправляемый процессу, когда срабатывает таймер, ранее установленный с помощью вызова **alarm**.

В оригинальном варианте ОС 4.3BSD и SVR3 определено 15 различных сигналов. Системы 4BSD и SVR4 поддерживают по 31 сигналу. Каждому из них присваивается номер от 1 до 31 (установка номера сигнала в 0 для различных функций имеет специальные значения, например "никаких сигналов"). Адресация сигналов по их номерам отличается в системах System V и BSD UNIX (например, **SIGSTOP** имеет номер 17 в 4.3BSD и номер 23 в SVR4). Более того, многие коммерческие реализации UNIX (такие как AIX) поддерживают больше чем 31 сигнал. Для идентификации сигналов в программе лучше использовать символические имена. Стандарт POSIX 1003.1 определяет символические имена для всех поддерживаемых им сигналов. Эти имена являются переносимыми, как минимум, для всех реализаций систем, совместимых со стандартом POSIX.

Обработка

Для каждого сигнала определено некоторое ***действие по умолчанию***, которое производится ядром системы. Всего таких действий пять:

- ***Аварийное завершение*** (**abort**). Завершает процесс после создания ***дампа состояния процесса*** (**core dump**), представляющего собой содержимое адресного пространства процесса и его контекст, записанные в файл **core**.
- ***Выход*** (**exit**). Завершает процесс без создания дампа состояния процесса.
- ***Игнорирование*** (**ignore**). Игнорирует сигнал.
- ***Остановка*** (**stop**). Приостанавливает процесс.
- ***Продолжение*** (**continue**). Возобновляет работу приостановленного процесса.

Процесс может переопределить действия, производимые по умолчанию для любого сигнала. Таким альтернативным вариантом может быть игнорирование сигнала или запуск определенной в приложении функции, называемой ***обработчиком сигнала***. Процесс может в любое время указать новое действие либо, наоборот, сбросить установки на действие по умолчанию. Процесс может временно блокировать сигнал (не поддерживается в SVR2 и более ранних версиях этой системы). В таком случае сигнал не будет доставлен до тех пор, пока не будет разблокирован.

При получении сигнала процесс может:

- ***игнорировать сигнал***;
- ***выполнить действия по умолчанию***;
- ***перехватить сигнал и самостоятельно его обработать***.

Текущее действие при получении сигнала называется ***диспозицией сигнала***.

Сигналы **`SIGKILL`** и **`SIGSTOP`** являются специальными, и приложения не могут:

- игнорировать;
- блокировать;
- или определять собственные обработчики для них.

Условия генерации сигнала и действие системы по умолчанию приведены в таблице (*Таблица 3.1*). Как видно из таблицы, при получении сигнала в большинстве случаев по умолчанию происходит завершение выполнения процесса. В ряде случаев в текущем рабочем каталоге процесса также создаётся файл core, в котором хранится образ памяти процесса. Этот файл может быть впоследствии проанализирован программой отладчиком для определения состояния процесса непосредственно перед завершением.

Что произойдет, если спящий процесс получит сигнал? Разбудят ли его для обработки этого сигнала или сигнал будет ожидать того момента, когда процесс проснется?

Система UNIX поддерживает два вида сна:

- ***прерываемый***;
- и ***непрерываемый***.

Процесс, находящийся в состоянии сна до события, которое может произойти в ближайшее время (например, завершение дискового ввода-вывода), пребывает в непрерываемом сне и не может быть разбужен поступающими сигналами.

Процесс, ожидающий такого события, как ввод-вывод терминала, который может не произойти в течение продолжительного времени, находится в прерываемом состоянии сна и будет разбужен посланным ему сигналом.

Таблица 3.1 Сигналы

SIGCHLD	Игнорировать	Сигнал, посылаемый родительскому процессу при завершении выполнения его потомка.
SIGINT	Завершить	Сигнал посылается ядром всем процессам текущей группы при нажатии клавиши прерывания (<Del> (или <Ctrl> + <C>)
SIGKILL	Завершить	Сигнал, при получении которого выполнение процесса завершается. Этот сигнал нельзя ни перехватить ни игнорировать.
SIGPIPE	Завершить	Сигнал посылается при попытке записи в канал или сокет, получатель данных которого завершил выполнение (закрыл соответствующий дескриптор).
SIGSTOP	Остановить	Сигнал отправляется всем процессам текущей группы при нажатии пользователем клавиш <Ctrl> + <Z>. Получение сигнала вызывает остановку выполнения процесса.
SIGTERM	Завершить	Сигнал обычно представляет своего рода предупреждение, что процесс, вскоре будет уничтожен. Этот сигнал позволяет процессу соответствующим образом "подготовиться к смерти"- удалить временные файлы, завершить необходимые транзакции и т. д. Команда kill(1) по умолчанию отправляет именно этот сигнал.
SIGUSR1	Завершить	Сигнал предназначен для прикладных задач как простейшее средство межпроцессорного взаимодействия.
SIGUSR2	Завершить	Сигнал предназначен для прикладных задач как простейшее средство межпроцессорного взаимодействия.

Ненадежные сигналы

Изначальная реализация сигналов (в системах SVR2 и более ранних версиях) была ***ненадежной*** и малоэффективной. И главным недостатком реализации как раз и была надежность доставки сигналов.

Рассмотрим следующий пример, пусть программист задал свой обработчик для сигнала **`SIGINT`**. После возникновения этого сигнала ядро системы сбросит действия, определенные для него, на установки по умолчанию. И, если нужно обработать также и все повторные сигналы, необходимо после каждого сигнала переустанавливать обработчик. Таким образом, обработчики сигналов не являются постоянно установленными и не маскируют повторения одного и того же сигнала.

Такой подход приводит к состязательности. Например, пользователь дважды быстро нажимает комбинацию клавиш Ctrl+C. Первое нажатие приводит к возникновению сигнала **`SIGINT`** и запуску пользовательского обработчика. Если повторное нажатие произошло до переустановки обработчика, ядро системы выполнит действие по умолчанию и завершит процесс. Такой подход приводит к существованию определенного промежутка времени между запуском и переустановкой обработчика, в течение которого сигнал не может быть перехвачен. Поэтому говорят, что ранние реализации UNIX имеют ***ненадежные сигналы***.

Если ядру системы необходимо послать сигнал процессу, находящемуся в состоянии прерываемого сна, оно не может знать, игнорирует процесс такой сигнал или нет. Таким образом, оно отправит сигнал и разбудит тем самым процесс, предполагая, что тот обрабатывает сигнал. Если процесс обнаружит, что он разбужен сигналом, который им игнорируется, он просто снова заснет. Такие лишние пробуждения приводят к совершенно ненужным переключениям контекста и потере времени на обработку сигналов. Было бы лучше, если бы ядро распознавало сигналы и сбрасывало те из них, которые должны игнорироваться, а сам процесс в этом не участвовал.

Надежные сигналы

Проблемы с ненадежными сигналами были решены в системе 4.2BSD, в которой был представлен механизм ***надежных сигналов***. Компания AT&T представила собственную версию надежных сигналов в своей системе SVR3. Эта версия была несовместима с интерфейсом BSD. Разработчики сохранили в реализации SVR3 совместимость с исходным механизмом сигналов, который был в SVR2. В обеих системах 4.2BSD и SVR3, была предпринята попытка решения одних и тех же проблем

разными способами. В итоге каждая из этих ОС обладает собственным набором системных вызовов, используемых для доступа к средствам управления сигналами. Эти вызовы имеют как различные имена, так и отличающуюся друг от друга семантику.

Стандарт POSIX.1 определил стандартный набор функций, которые должны быть реализованы во всех системах, удовлетворяющих этому стандарту. Функции согласно этому стандарту могут быть реализованы:

- как в виде системных вызовов,
- так и в виде библиотечных процедур.

На основе требований POSIX в системе SVR4 был представлен новый интерфейс, который удовлетворял стандарту POSIX и оказался, совместим с BSD и со всеми предыдущими версиями UNIX, созданными ранее компанией AT&T.

Все реализации механизма надежных сигналов обладают общими возможностями:

- ***Постоянно установленные обработчики*** (***persistent handlers***). Обработчики сигналов остаются установленными даже после возникновения сигнала и не требуют переустановок. Не существует временного интервала между запуском обработчика сигнала и его переустановкой, во время которого повторно поступивший сигнал может, например, завершить процесс.
- ***Маскирование*** (***masking***). Сигнал может быть временно маскирован (блокирован). Если вырабатывается сигнал, который уже блокирован процессом, ядро системы будет помнить об этом и не станет посылать такой сигнал процессу незамедлительно. Сигнал будет переправлен и обработан после того, как процесс разблокирует его. Такой подход лает возможность программисту защитить ***критические области кода*** от прерывания его выполнения при возникновении определенных сигналов.
- ***Спящие процессы*** (***sleeping process***) Некоторая информация о диспозиции сигналов в процессе является видимой для ядра (посредством хранения данных в структуре proc вместо области u), даже если процесс не находится в текущий момент на выполнении. Следовательно, если ядро генерирует сигнал для процесса, находящегося в прерываемом сне, но тот игнорирует или блокирует данный сигнал, то ядро не станет будить такой процесс.
- ***Разблокирование и ожидание*** (***unblock and wait***). Системный вызов **`pause`** блокирует процесс до тех пор, пока ему не будет доставлен сигнал. Механизм надежных сигналов предлагает еще один вызов,

sigpause, который атомарно демаскирует сигнал и блокирует процесс до тех пор, пока тот не получит такой сигнал. Если демаскированный сигнал уже находится в ожидании, то произойдет немедленный возврат из этого системного вызова.

Неименованные и именованные каналы

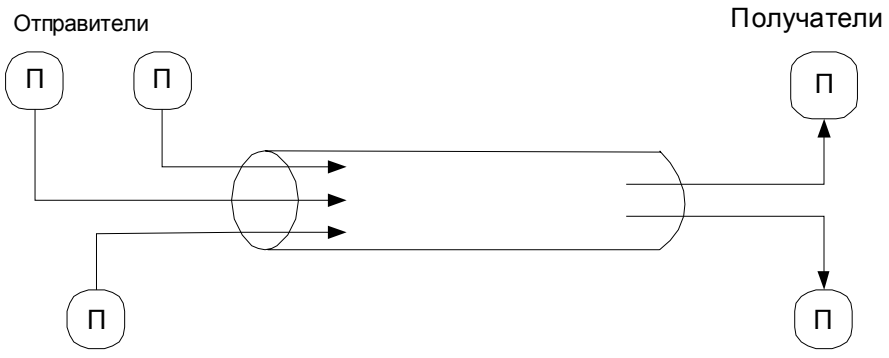
Неименованные каналы (pipe)

Неименованные каналы - это одна из первых форм IPC в UNIX, которая появилась в 1973 году в третьей версии (Third Edition). Главный недостаток неименованных каналов это отсутствие имени, поэтому их могут использовать для взаимодействия только родственные процессы.

В традиционных реализациях UNIX ***программные каналы*** (***pipe***) представляют собой однонаправленный неструктурированный поток данных фиксированного размера, работающий по принципу FIFO ("первым вошел, первым вышел"). Отправители добавляют данные в конец канала, получатели извлекают их из его начала. После того как данные прочитаны, они сразу же удаляются из канала и больше недоступны другим получателям.

- Процесс, который попытается прочитать данные из пустого канала, будет приостановлен до тех пор, пока в канале не появятся данные.
- Если процесс попытается записать в заполненный канал, то он будет приостановлен до того момента, пока другой процесс не прочтет данные из канала, и в нем появится свободное место.

Рисунок 3.1 Использование канала



Для создания программного канала применяется системный вызов **pipe**, который возвращает два дескриптора файла:

- один для чтения;
- и один для записи.

Эти дескрипторы наследуются процессами потомками, которые таким образом могут получить доступ к каналу. Каждый процесс, может, как читать, так и записывать информацию, а так же делать и то и другое (см. *Рисунок 3.1*). Однако в большинстве случаев программный канал используется двумя процессами, на двух его концах.

Операции ввода-вывода над каналами аналогичны операции над файлами. Они производятся через системные вызовы **`read`** и **`write`** с помощью ***дескриптора канала***. Процесс иногда может и не знать о том, что работает с каналом, а не с обычным файлом.

Командные интерпретаторы управляют программными каналами, считая, что имеется только один отправитель (записывающий данные), тем самым образуется однонаправленный поток данных. Наиболее общим применением программного канала является перенаправление вывода одной программы на вход другой. Для этого пользователи объединяют две программы в конвейер при помощи оператора |.

С точки зрения IPC программные каналы являются эффективным способом передачи данных от одного процесса другому. Однако они обладают несколькими ограничениями:

- так как чтение данных из канала приводит к их удалению, он не может быть использован для широковещательной передачи информации нескольким адресатам;
- данные канала интерпретируются как поток байтов, границы сообщения заранее не известны. Если отправитель посылает через канал несколько объектов различной длины, получатель не может определять, сколько объектов ему было передано. Кроме того, он не знает, где заканчивается один объект и начинается другой;
- если данные из канала считывают несколько процессов, отправитель не в состоянии передать данные какому-то определенному получателю. Точно так же при наличии нескольких отправителей не существует способа определения, какой из них отослал данные.

Именованные каналы (named pipe)

Реализация программных каналов может быть осуществлена различными способами. В традиционных вариантах (например, в SVR2) для этого применяются механизмы файловой системы и ассоциации ***индексного дескриптора файла*** (***inode***) и элемента таблицы файлов для каждого канала.

В ОС System V UNIX и других коммерческих вариантах UNIX представлены файлы FIFO, называемые также ***именованными каналами*** (***named pipe***). Они отличаются от обычных неименованных каналов способами создания и доступа. Пользователь создает файл FIFO при помощи системного вызова **`mknod`**, передавая ему имя файла и режим его создания. Поле режима описывает тип файла **`S_IFIFO`** и права доступа. После этого процесс, обладающий соответствующими полномочиями, может открывать файл FIFO и выполнять операции чтения и записи. Семантика чтения/записи для файлов FIFO похожа на аналогичные операции над программными неименованными каналами.

Файлы FIFO обладают преимуществами по сравнению с программными каналами:

- они имеют имя в пространстве имен файловой системы;
- они доступны любым процессам (неродственным);
- они являются постоянными, а не существуют только во время выполнения процессов.

Недостаток файлов FIFO в том, что по завершении использования их необходимо принудительно удалять.

ЛАБОРАТОРНАЯ РАБОТА 3.1

ИСПОЛЬЗОВАНИЕ СИГНАЛОВ (IPC)

Теория

Сигналы являются способом передачи от одного процесса другому или от ядра какому-либо процессу уведомления о возникновении определенного события. Их можно рассматривать как простейшую форму межпроцессного взаимодействия (IPC). Сигналы обеспечивают механизм вызова определенной процедуры при наступлении некоторого события. Каждое событие имеет свой идентификатор и соответствующую ему символьную константу. Например, сигнал прерывания, посылаемый процессу при нажатии пользователем клавиши <Del> или <Ctrl+C> (см. **stty** -a), имеет имя **SIGINT**.

Для отправления сигнала служит команда **kil**l:

```
kill sig_no pid
```

где **sig_no** - номер или символическое название сигнала, а **pid** - идентификатор процесса, которому посылается сигнал.

Администратор системы может посылать сигналы любым процессам, обычный же пользователь может посылать сигналы любым процессам, владельцем которых он является. Например, запустим процесс в фоновом режиме и пошлем ему сигнал завершения выполнения **SIGTERM**:

```
our_program& //запустили программу в фоновом режиме
kill $!
```

По умолчанию команда **kill** посылает сигнал **SIGTERM**; переменная **$!** содержит **PID** последнего процесса, запущенного в фоновом режиме.

При получении сигнала процесс может:

- игнорировать сигнал. Не следует игнорировать сигналы, вызванные аппаратной частью (например, деление на 0 или ссылка на недопустимые области памяти), так как дальнейшее поведение процесса непредсказуемо;
- выполнить действие по умолчанию. Обычно это приводит к завершению выполнения процесса;
- перехватить сигнал и самостоятельно обработать его.

Помните, что сигналы **SIGKILL** и **SIGSTOP** нельзя ни перехватить, ни игнорировать.

По умолчанию команда **kill** посылает сигнал с номером 15 (**SIGTERM**), для которого действие по умолчанию - завершение выполнения процесса. Иногда процесс продолжает существовать, и после отправления сигнала **SIGTERM**. В этом случае можно послать процессу сигнал **SIGKILL (9)**, поскольку этот сигнал нельзя ни перехватить, ни игнорировать:

```
kill -9 pid
```

Возможны ситуации, когда процесс не исчезает и в этом случае. Это может произойти для следующих процессов:

- ***Процессы-зомби***. Фактически процесс не существует, осталась лишь запись в системной таблице процессов, поэтому удалить его можно только перезапуском ОС. Зомби в небольших количествах не опасны, но если их много, то может переполниться таблица процессов;
- ***Процессы, ожидающие недоступные ресурсы NFS*** (Network File System), например, записывающие данные в файл файловой системы удаленного компьютера, отключившегося от сети. Эту ситуацию можно преодолеть, послав процессу сигнал **SIGINT** или **SIGQUIT**;
- Процессы, ожидающие завершения операции с устройством.

Теперь рассмотрим системный вызов **kill**:

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

Аргумент **pid** адресует процесс, которому посылается сигнал. Аргумент **sig** задает отправляемый сигнал. С помощью системного вызова **kill** процесс может послать сигнал как самому себе, так и другому процессу. При этом процесс, посылающий сигнал, должен иметь те же реальный и эффективный идентификаторы, что и процесс, которому сигнал отправляется. Конечно, процессы администратора (**root**) могут отправлять сигналы любым процессам системы.

Ненадежные сигналы

Рассмотрим функцию **signal**. Эта функция позволяет изменить диспозицию сигнала, которая по умолчанию устанавливается ядром UNIX. Порожденный вызовом **fork** процесс наследует диспозицию сигналов от своего родителя. Но при вызове **exec** диспозиция всех перехватываемых сигналов будет установлена на действие по

умолчанию. Это нормально, поскольку образ новой программы не содержит функции обработчика, определённой диспозицией сигнала перед вызовом **exec**. Рассмотрим функцию **signal**:

```
#include <signal.h>

void (*signal(int sig, void (*disp) (init))) (int);
```

Аргумент **sig** определяет сигнал, диспозицию которого нужно изменить. Аргумент **disp** определяет новую диспозицию сигнала, которой может быть определенная пользователем функция-обработчик или одно из следующих значений:

- **SIG_DFL -** Указывает ядру, что при получении процессом сигнала необходимо вызвать системный обработчик, т. е. выполнить действие по умолчанию;
- **SIG_IGN** - Указывает, что сигнал следует игнорировать. Помните, что не все сигналы можно игнорировать.

В случае успешного завершения **signal** возвращает предыдущую диспозицию. Это может быть функция-обработчик сигнала или системные значения **SIG_DFL** или **SIG_IGN**. Возвращаемое значение может быть использовано для восстановления диспозиции в случае необходимости.

Использование функции **signal** - это работа с устаревшими или ненадежными сигналами. Существуют следующие проблемы:

- процесс не может заблокировать сигнал, т. е. отложить получение сигнала на период выполнения критического участка кода;
- каждый раз при получении сигнала его диспозиция устанавливается на действие по умолчанию.

Данную функцию сохранили, чтобы поддерживать старые версии приложений.

Рассмотрим пример:

```
#include <signal.h>

/*Функция-обработчик сигнала*/
static void sig_hndl(int sig)
{
/*Восстановим диспозицию*/
signal(SIGINT, sig_hndl);
printf ("Получен сигнал SIGINT\n");
}
```

```
main (.)
{
/*Установим диспозицию*/
signal(SIGINT, sig_hndl);
signal(SIGUSR1, SIG_DFL);
signal(SIGUSR2, SIG_IGN);
while(1)
pause();
}
```

В этом примере изменена диспозиция трех сигналов (**SIGINT**, **SIGUSR1** и **SIGUSR2**):

- при получении сигнала **SIGINT** вызывается обработчик **sig_hndl()**;
- при получении сигнала **SIGUSR1** производится действие по умолчанию (процесс завершает работу);
- а сигнал **SIGUSR2** игнорируется.

После установки диспозиции сигналов в бесконечном цикле вызывается функция **pause**. Каждый раз при получении сигнала **SIGINT** нужно восстанавливать требуемую диспозицию, иначе получение следующего сигнала **SIGINT** вызвало бы завершение процесса (действие по умолчанию).

Надежные сигналы

Стандарт POSIX.1 определил новый набор функций управления сигналами, основанный на интерфейсе 4.2BSD UNIX.

В модели сигналов POSIX используется понятие ***набора сигналов*** (***signal set***), который описывается переменной типа **sigset_t**. Каждый бит этой переменной отвечает за один сигнал. Во многих системах тип **sigset_t** имеет длину 32 бита, ограничивая количество возможных сигналов числом 32.

Рассмотрим функции, которые позволяют управлять наборами сигналов:

```
# include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(sigset_t *set, int signo);
```

В отличие от функции **signal**, изменяющей диспозицию сигналов, данные функции позволяют модифицировать структуру данных **sigset_t**, определенную процессом. Для управления, непосредственно сигналами используются дополнительные функции, которые рассмотрим позже.

Функция **sigemptyset** инициализирует набор, очищая все биты. Если процесс вызывает **sigfillset**, то набор будет включать все сигналы, известные системе. Функции **sigaddset** и **sigdelset** позволяют добавлять или удалять сигналы набора. Функция **sigismember** позволяет проверить, входит ли указанный параметром **signo** сигнал в набор.

Вместо функции **signal** стандарт POSIX.1 определяет функцию **sigaction**, позволяющую установить диспозицию сигналов, узнать ее текущее значение или сделать и то и другое одновременно.

```
# include <signal.h>

int sigaction(int sig, const struct sigaction *act, struct sigaction *oact);
```

Вся необходимая для управления сигналами информация передается через указатель на структуру **sigaction**.

```
struct sigaction {
void (*sa_handler) ( ) //Обработчик сигнала sig
void (*sa_sigaction) (int, siginfo_t *, void *)
//Обработчик сигнала sig при установленном флаге SA_SIGINFO
sigset_t sa mask //Маска сигналов
int sa_flags //Флаги
};
```

Поле **sa_handler** определяет действие, которое необходимо предпринять при получении сигналов, и может принимать значения **SIG_IGN**, **SIG_DEL** или адрес функции обработчика. Если значение **sa_handler** или **sa_sigaction** не равны **NULL**, то в поле **sa_mask** передается набор сигналов, которые будут добавлены к маске сигналов перед вызовом обработчика. Каждый процесс имеет установленную маску сигналов, определяющую сигналы, доставка которых должна быть заблокирована. Если определенный бит маски установлен, соответствующий ему сигнал будет заблокирован. После возврата из функции обработчика значение маски возвращается к исходному значению. Сигнал, для которого установлена функция обработчик, также будет заблокирован перед ее вызовом. Такой подход гарантирует, что во время обработки, последующее поступление определенных сигналов будет приостановлено до завершения функции. Как правило, UNIX не поддерживает очередей сигналов, и это значит, что блокировка нескольких однотипных сигналов в конечном итоге вызовет доставку лишь одного.

В системах UNIX BSD4.x структура sigaction имеет следующий вид:

```
struct sigaction {
void (*sa_handler) ( );
sigset_t sa_mask;
int sa_flags;
};
```

где функция-обработчик определена следующим образом:

```
void handler (int signo, int code, struct sigcontext scp);
```

В первом аргументе **signo** содержится номер сигнала, **code** определяет дополнительную информацию о причине поступления сигнала, а **scp** указывает на контекст процесса.

Чему нужно научиться?

Изучить системные вызовы **kill**, **signal** *и научиться использовать надежные и ненадежные сигналы для реализации межпроцессного взаимодействия (IPC).*

Задания A (ненадежные сигналы), B (надежные сигналы)

Уровень 1 (A)

Использовать команду **kill -l**, чтобы вывести список используемых сигналов.

Написать программу father.c:

- **1-ый шаг.** Процесс-родитель (**father**) порождает процессы-сыновья (**son1**, **son2**, **son3**);
- **2-ой шаг.** Процесс-родитель (**father**) посылает сигнал **SIGUSR1** каждому процессу сыну;
- **3-ий шаг.** Обеспечить в процессах-сыновьях следующую реакцию на сигнал **SIGUSR1**:
  - **son1** должен обеспечить реакцию на этот сигнал по умолчанию;
  - **son2** должен игнорировать этот сигнал;
  - **son3** должен перехватить и самостоятельно обработать этот сигнал.

Проанализировать таблицу процессов до и после посылки сигналов с помощью системного вызова **system("ps -s >> file")**.

Уровень 2 (А)

Написать программу sig_father.c, в которой изменена диспозиция сигналов:

- сигналы **SIGUSR1** и **SIGUSR2** обработать самим (написать свои обработчики);
- сигнал **SIGINT** обрабатывается по умолчанию;
- сигнал **SIGCHLD** игнорируется.

Породить процесс-сын **sig_son** и ждать прихода сигналов.

В своих обработчиках вы должны восстанавливать диспозицию сигналов. Кроме того, на экран должно выводиться информация о том, получен сигнал или нет.

Процесс-сын sig_son.c должен:

- получить идентификатор родительского процесса;
- послать процессу отцу сигнал **SIGUSR1**;
- и проверить удачно или неудачно отправлен указанный сигнал.

Попробуйте также посылать сигналы из командной строки. Для этого запустите на выполнение исполняемый модуль **./sig_father&** и сигналы посылайте с терминала, используя **kill -s <#сигнала> <PID sig_father>**.

Откомпилируйте обе программы:

- **gcc -o sig_father sig_father.c**;
- **gcc -o sig_son sig_son.c**.

Уровень 3 (А)

Используйте программу, созданную для задания В (*“Уровень 1, 2, 3 (В)” on page 66*), и помните, что для компиляции вашего файла надо использовать команду **gcc -o OUTPUT /lib/libthread.so.0 INPUT.C**

Сначала попробуйте удалить нить, используя ее идентификатор, командой **kill**.

Затем измените программу так, чтобы:

- из первой нити посылался сигнал **SIGUSR1** во вторую нить;

- затем удалялась бы вторая нить. Для этого используйте функцию **pthread_kill(t2, SIGUSR1)**, где **t2** - дескриптор второй нити.

Еще раз измените программу:

- создайте собственный обработчик сигнала **SIGUSR1** для второй нити. В нем просто печатайте сообщение о том, что пришел сигнал;
- для выхода из обработчика сигнала используйте функцию **pthread_exit(NULL)**.

Уровень 1, 2, 3 (В)

Напишите программу sigact.c, позволяющую заблокировать сигнал **SIGINT**.

Вся необходимая для управления сигналами информация должна передаваться через указатель на структуру **sigaction**.

```
void (*mysig (int sig, void (*handler) (int))) (int) //надежная обработка
сигналов
{
   struct sigaction act;
   act.sa_handler = handler;//установка обработчика сигнала № sig
   sigemptyset(&act.sa_mask);//обнуление маски
   sigaddset(&act.sa_mask, SIGINT);//блокировка сигнала SIGINT
   act.sa_flags = 0;
   if(sigaction(sig, &act, 0) < 0)
      return (SIG_ERR);
   return (act.sa_handler);
}
```

Блокировку реализуйте, вызвав "засыпание" процесса на одну минуту из обработчика пользовательских сигналов. В основной программе установите диспозицию этих сигналов.

Откомпилируйте программу (**gcc -o sigact sigact.c**) и запустите на выполнение исполняемый модуль **./sigact&**. С терминала отправьте процессу **sigact** сигнал **SIGUSR1** или **SIGUSR2**, а затем сигнал **SIGINT**. Измените обработчик так, чтобы отправка сигнала **SIGINT** производилась из обработчика функцией **kill(SIGINT, getpid())**, установив в основной программе диспозицию:

```
struct sigaction act, oldact;
. . .
if(sigaction(sig, &act, &oldact) < 0)
   return(SIG_ERR);
   return(oldact.sa_handler);
```

Сравните обработчики надежных и ненадежных сигналов.

Скелет кода для задания А

sig_father.c

```
#include <stdio.h>
#include <signal.h>

void restore_signals(void)
{
  signal(SIGUSR1,SIG_DFL);
  signal(SIGUSR2,SIG_DFL);
  signal(SIGINT,SIG_DFL);
  signal(SIGCHLD,SIG_DFL);
}

void sig_handler_usr1(int sig)
{
  printf("usr1 handler %i\n",sig);
  if(sig==SIGUSR1)
  {
    printf("Our signal!\n");
  }
  else
  {
    printf("Not our signal\n");
  }
  printf("pid=%i ppid=%i\n",getpid(),getppid());
  restore_signals();
}

void sig_handler_usr2(int sig)
{
  printf("usr2 handler %i\n",sig);
  if(sig==SIGUSR2)
  {
    printf("Our signal!\n");
  }
  else
  {
    printf("Not our signal\n");
  }
  printf("pid=%i ppid=%i\n",getpid(),getppid());
  restore_signals();
}

int main()
{
  int son_pid, status, i;
  printf("sig_father is starting! pid=%i\n",getpid());
  signal(SIGUSR1,sig_handler_usr1);
  signal(SIGUSR2,sig_handler_usr2);
  signal(SIGINT,SIG_DFL);
  signal(SIGCHLD,SIG_IGN);

  if((son_pid=fork())==0)
  {
```

```
    exec("sig_son","sig_son",NULL);
  }
  sleep(1000);
  wait(&status);
  return 0;
}
```

sig_son.c

```
#include <stdio.h>
#include <signal.h>

int main(void)
{
  int son_pid,status;

  printf("sig_son is starting!");
  printf("pid=%i ppid=%i\n",getpid(),getppid());

  if((kill(getppid(),SIGUSR1))==-1)
  {
    printf("Send signal with Error!\n");
  }
  else
  {
    printf("Send signal to father successfully!\n");
  }
  return 0;
}
```

Скелет кода для задания B

sigact.c

```
#include <stdio.h>
#include <signal.h>

void sig_handler(int sig)
{
  printf("Signal %i is comming\n",sig);
  sleep(60);
}

void set_action(int sig,struct sigaction* new_action, struct sigaction*
old_action)
{
  new_action->sa_handler=sig_handler;
  sigemptyset(&(new_action->sa_mask));
  sigaddset(&(new_action->sa_mask),SIGINT);
  new_action->sa_flags=0;

  if(sigaction(sig,new_action,old_action)<0)
  {
    printf("Action is not setted\n");
```

```
    exit(0);
  }
  printf("Action is setted\n");
  return;
}

void restore_action(int sig, struct sigaction* old_action)
{
  if(sigaction(sig,old_action,NULL)<0)
  {
    printf("We can not restore action!\n");
    exit(0);
  }
  printf("We restored action!\n");
  return;
}

int main(void)
{
  struct sigaction new_usr1_action,old_usr1_action,
new_usr2_action,old_usr2_action;

  printf("Sigact pid=%i\n",getpid());

  set_action(SIGUSR1,&new_usr1_action,&old_usr1_action);
  set_action(SIGUSR2,&new_usr2_action,&old_usr2_action);

  if((kill(getpid(),SIGUSR1))==-1)
  {
    printf("Signal send with error!\n");
  }
  else
  {
    printf("Signal send OK!\n");
  }

  sleep(60);

  restore_action(SIGUSR1,&old_usr1_action);
  restore_action(SIGUSR2,&old_usr2_action);

  return 0;
}
```

ЛАБОРАТОРНАЯ РАБОТА 3.2

ИСПОЛЬЗОВАНИЕ КАНАЛОВ (IPC)

Теория

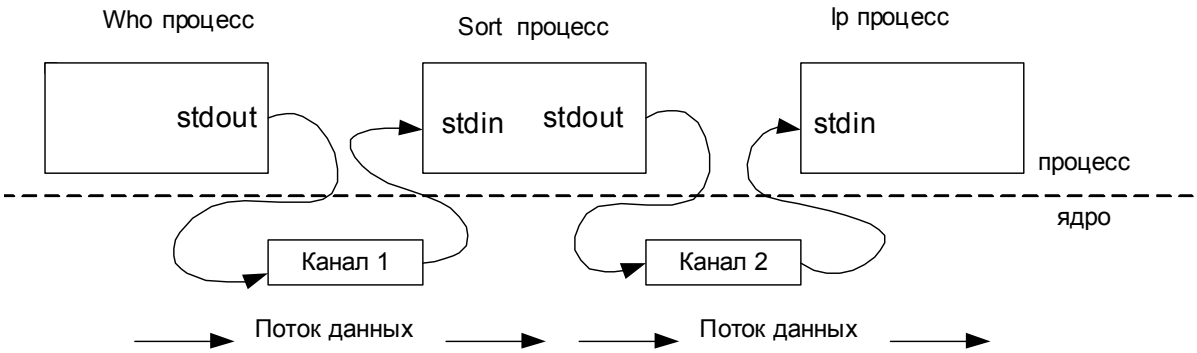
Неименованные каналы

Рассмотрим команду:

```
who | sort | lp
```

Интерпретатор (shell) создает три процесса с двумя каналами между ними. Интерпретатор также подключает открытый для чтения конец каждого канала к стандартному потоку ввода, а открытый на запись - к стандартному потоку вывода (см. *Рисунок 3.2*).

Рисунок 3.2 Каналы между тремя процессами при конвейерной обработке



Таким образом, три процесса обменялись данными. При этом использовались программные каналы, обеспечивающие однонаправленную передачу данных между тремя процессами.

Для создания канала используется системный вызов **pipe**:

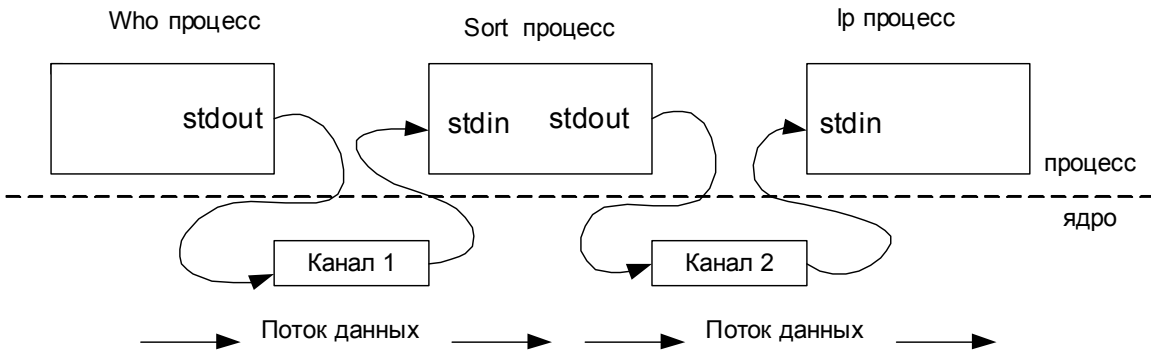
```
# include <unistd.h>

int pipe(int fd[2]);
```

Функция возвращает два файловых дескриптора:

- fd[0], который открыт для чтения из канала;
- fd[1], который открыт для записи в канал.

На рисуеке (*Рисунок 3.3*) изображен канал при использовании его единственным процессом. Теперь, если один процесс записывает данные в fd[1], другой сможет получить эти данные из fd[0].

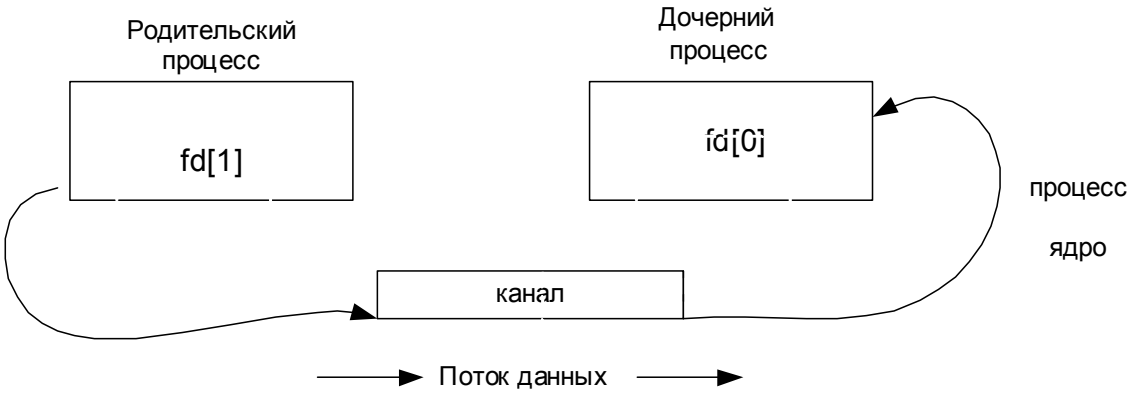
Рисунок 3.3 Канал в процессе



Хотя канал создается одним процессом, он редко используется только этим процессом. Каналы обычно используются для связи между процессами (родительским и дочерним) следующим образом:

- процесс создает канал;
- затем вызывает **`fork`**, чтобы создать свою копию - дочерний процесс;
- затем родительский процесс закрывает открытый для чтения конец канала, а дочерний открытый на запись конец канала остается. Это обеспечивает одностороннюю передачу данных между процессами (см. *Рисунок 3.4*).

Рисунок 3.4 Канал между двумя процессами



Вопрос заключается только в том, как другой процесс сможет получить свой файловый дескриптор?

Вспомним наследуемые атрибуты при создании процесса. Дочерний процесс наследует и разделяет все файловые дескрипторы родительского процесса. То есть доступ к дескрипторам fd канала могут получить сам процесс, вызвавший **pipe**, и его дочерние процессы. В этом заключается серьезный недостаток каналов, поскольку они могут быть использованы для передачи данных только между родственными процессами и не могут использоваться в качестве средства межпроцессорного взаимодействия между независимыми процессами.

Хотя в приведенном примере (см *Рисунок 3.2*) может показаться, что процессы **who**, **sort** и **lp** независимы, на самом деле эти процесс создаются процессом **shell** и являются родственными.

Именованные каналы (FIFO)

Название каналов **FIFO** происходит от выражения **First In First Out** (***первый вошел - первый вышел***). FIFO очень похожи на каналы, поскольку являются однонаправленным средством передачи данных, причем чтение данных происходит в порядке их записи. Однако в отличие от программных каналов, FIFO имеют имена, которые позволяют независимым процессам получить к этим объектам доступ. Поэтому иногда FIFO также называют именованными каналами. FIFO являются средством UNIX System V и не используются в BSD. Впервые FIFO были представлены в System III, однако они до сих пор не документированы и поэтому мало используются.

FIFO является отдельным типом файлов в файловой системе UNIX (**ls -l** покажет символ **p** в первой позиции). Для создания FIFO используется системный вызов **mknod**:

```
int mknod(char *pathname, int mode, int dev);
```

- где **pathname** - имя файла в файловой системе (имя FIFO);
- **mode** - флаги владения, права доступа и т.д.;
- **dev** - при создании FIFO игнорируется.

После создания канал FIFO может быть открыт на запись и на чтение, причем запись и чтение могут происходить в разных независимых процессах.

Еще раз повторим правила, по которым работают каналы FIFO и обычные каналы и которые надо обязательно учитывать при программировании:

- При чтении меньшего числа байтов, чем находится в канале или FIFO, возвращается требуемое число байтов, остаток сохраняется для последующих чтений.
- При чтении большего числа байтов, чем находится в канале или FIFO, возвращается доступное число байтов. Процесс, читающий из канала, должен соответствующим образом обработать ситуацию, когда прочитано меньше, чем заказано.
- Если канал пуст и ни один процесс не открыл его на запись, при чтении из канала будет получено 0 байтов. Если один или более процессов открыли канал для записи, вызов **`read`** будет заблокирован до появления данных (если для канала или FIFO не установлен флаг отсутствия блокировки **`O_NDELAY`**).
- Запись числа байтов, меньшего емкости канала или FIFO, гарантировано атомарно. Это означает, что в случае, когда несколько процессов одновременно записывают в канал, порции данных от этих процессов не перемешиваются.
- При записи большего числа байтов, чем это позволяет канал или FIFO, вызов write блокируется до освобождения требуемого места. При этом атомарность операции не гарантируется. Если процесс пытается записать данные в канал, не открытый ни одним из процессов на чтение, процессу генерируется сигнал **`SIGPIPE`**, а вызов **`write`** возвращает **0** с установкой ошибки (**`errno=EPIPE`**) (если процесс не установил обработки сигнала **`SIGPIPE`**, производится обработка по умолчанию - процесс завершается).

Чему нужно научиться?

Научиться использовать каналы и для реализации межпроцессного взаимодействия (IPC).

Задания A (каналы), B (FIFO)

Уровень 1 (A)

Посмотрите на Скелет кода для задания 1 (A) (*"Скелет кода для задания 1 (A)" on page 96*) и **разберитесь, что там происходит**.

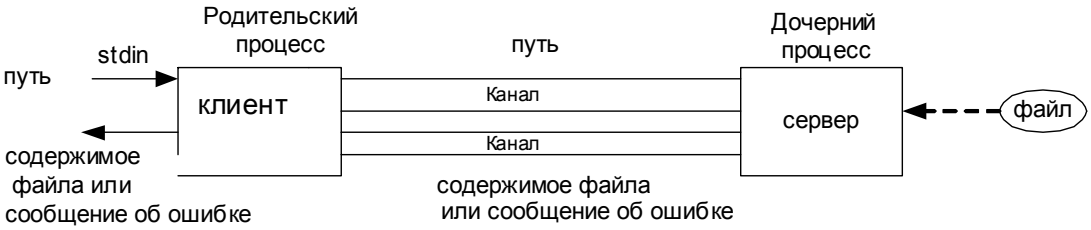
Уровень 2, 3 (A)

Давайте разработаем клиент-серверное приложение (см. *Рисунок 3.5*):

- Клиент будет считывать полное имя файла из стандартного потока ввода и записывать его в канал;

- Сервер будет считывать это имя из канала и пытаться открыть файл, используя это имя;
  - Если файл успешно откроется, то сервер будет передавать его содержимое в канал. В противном случае он вернет клиенту сообщение об ошибке.
- Клиент будет считывать данные из канала и записывать их в стандартный поток вывода.
  - Клиент либо выведет содержимое файла, либо сообщение об ошибке.

Рисунок 3.5 Приложение клиент-сервер



Уровень 1 (B)

Посмотрите на и разберитесь, что там происходит.

Уровень 2 (B)

Используйте именованные каналы для того, чтобы сделать клиент и сервер из задания Уровень 2, 3 (A) (*“Уровень 2, 3 (A)” on page 95*) неродственными процессами.

Уровень 3 (B)

А теперь измените неродственные клиент и сервер так, чтобы они могли обмениваться структурированными данными.

Скелет кода для задания 1 (A)

```
#include <stdio.h>

int main(void)
{
  int filedes[2];// 0 - read; 1 - write;
  char temp_ch;
  FILE* fout;

  if(pipe(filedes)<0)
  {
    printf("Father can not create Pipe!\n");
    exit(0);
```

```
  }
  printf("Father created Pipe!\n");

  if(fork()==0)
  {
    FILE* fin;
    char ch;
    close(filedes[0]);
    printf("Child is working!\n");
    fin=fopen("input.txt","rt");
    while(fscanf(fin,"%c",&ch)==1)
    {
      write(filedes[1],&ch,1);
    }
    fclose(fin);
    close(filedes[1]);
    printf("End of Child!\n");
    exit(0);
  }
  else
  {
    close(filedes[1]);
    printf("Father is working again!\n");
    fout=fopen("output.txt","wt");
    while(read(filedes[0],&temp_ch,1))
    {
      printf("%c",temp_ch);
      fprintf(fout,"%c",temp_ch);
    }
    fclose(fout);
    printf("End of Father!\n");
  }
  return;
}
```

Скелет кода для задания 2, 3 (A)

```
…
main(int argc, char **argv)
{
  int pipe1[2], pipe2[2];
  pid_t childpid;

  pipe(pipe1);
  pipe(pipe2);

  if((childpid=fork())==0)
  {// child
    close(pipe1[1]);
    close(pipe2[0]);

    server(pipe1[0],pipe2[1]);
    exit(0);
  }
  // parent
```

```
  close(pipe1[0]);
  close(pipe2[1]);

  client(pipe2[0],pipe1[1]);

  waitpid(childpid,NULL,0);
  exit(0);
}

void client(int readfd, int writefd)
{
  size_t len;
  ssize_t n;
  char buff[MAXLINE];

  fgets(buff,MAXLINE,stdin);
  len=strlen(buff);
  if(buff[len-1]=='\n')
  len--;
  write(writefd,buff,len);
  while((n=read(readfd,buff,MAXLINE))>0)
    write(…);
}

void server(int readfd,writefd)
{
  int fd;
  ssize_t n;
  char buff[MAXLINE+1];

  if((n=read(readfd,buff,MAXLINE))==0)
  …
  buff[n]='\0';
  if((fd=open(buff,O_RDONLY))<0)
  {
    …
    n=strlen(buff);
    write(writefd,buff,n);
  }
  else
  {
    while((n=read(fd,buff,MAXLINE))>0)
      write(writefd,buff,n);
    close(fd);
  }
}
```

Скелет кода для задания 1 (В)

Server.c

```
#include <sys/types.h>
#include <sys/start.h>
#define FIFO "fifo/1"
#define MAXBUFF 80

main()
{
int readfd, n;
char buff[MAXBUFF]; /*буфер для чтения данных из FIFO*/
/*Создадим специальный файл FIFO с открытыми для всех правами доступа на
чтение и запись*/
if(mknode(FIFO, S_IFIFO | 0666, 0) < 0){
printf("Невозможно создать FIFO\n"); exit(1);}
/*Получим доступ к FIFO*/
if((readfd = open(FIFO, O_RDONLY)) < 0){
printf("Невозможно открыть FIFO\n"); exit(1);}
/*Прочитаем сообщение ("Здравствуй, Мир!") и выведем его на экран*/
while((n = read(readfd, buff, MAXBUFF)) > 0)
if(write(1, buff, n) != n){
printf("Ошибка вывода\n"); exit(1);}
/*Закроем FIFO, удаление FIFO - дело клиента*/
close(readfd);
exit(0);
}
```

Client.c

```
#include <sys/types.h>
#include <sys/start.h>
/*Соглашение об имени FIFO*/
#define FIFO "fifo/1"

main()
{
int writefd, n;
/*Получим доступ к FIFO*/
if((writefd = open(FIFO, O_WRONLY)) < 0){
printf("Невозможно открыть FIFO\n"); exit(1); }
/*Передадим сообщение серверу FIFO*/
if(write(writefd, "Здравствуй, Мир!\n", 18) != 18){
printf("Ошибка записи\n"); exit(1); }
/*Закроем FIFO*/
close(writefd);
/*Удалим FIFO*/
if(unlink(FIFO) < 0){
printf("Невозможно удалить FIFO\n"); exit(1); }
exit(0); }
```

Скелет кода для задания 2 (B)

Client_fifo1.c

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"
#define MAXLINE 4096

void client(int readfd, int writefd);

int main(int argc,char **argv)
{
        int readfd, writefd;

        if( (writefd=open(FIFO1, O_WRONLY))<0)
        {
          printf("Client: can not open FIFO1 for write\n");
                exit(1);
        }
  printf("Client: FIFO1 is opened for write writefd=%d\n",writefd);

        if( (readfd=open(FIFO2, O_RDONLY))<0)
        {
          printf("Client: can not open FIFO2 for read\n");
                exit(1);
        }
  printf("Client: FIFO2 is opened for read readfd=%d\n",readfd);

        client(readfd,writefd);

        close(readfd);
        close(writefd);

        if (unlink(FIFO1) < 0)
        {
           printf("Client: can delete FIFO1\n");
                    exit(1);
        }
  printf("Client: FIFO1 is deleted!\n");

        if (unlink(FIFO2) < 0)
        {
           printf("Client: can delete FIFO2\n");
                    exit(1);
        }
  printf("Client: FIFO2 is deleted!\n");

  printf("Client is terminated!\n");

        exit(0);
}
```

```
void client(int readfd, int writefd)
{
…
}
```

Server_fifo1.c

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"
#define MAXLINE 4096

void server(int readfd, int writefd);

int main(int argc,char **argv)
{
        int readfd, writefd;

        if( mknod(FIFO1, S_IFIFO | 0666, 0)<0)
        {
                printf("Server: can not create FIFO1\n");
                exit(1);
        }
  printf("Server: FIFO1 is created!\n");
        if( mknod(FIFO2, S_IFIFO | 0666, 0)<0)
        {
                unlink(FIFO1);
                printf("Server: can not create FIFO2\n");
                exit(1);
        }
  printf("Server: FIFO2 is created!\n");

        if( (readfd=open(FIFO1, O_RDONLY))<0)
        {
          printf("Server: can not open FIFO1 for read\n");
                exit(1);
        }
  printf("Server: FIFO1 is opened for read and readfd=%d\n",readfd);

        if( (writefd=open(FIFO2, O_WRONLY))<0)
        {
          printf("Server: can not open FIFO2 for write\n");
                exit(1);
        }
  printf("Server: FIFO2 is opened for write and writefd=%d\n",writefd);

        server(readfd,writefd);

        exit(0);
}
```

```
void server(int readfd, int writefd)
{
…
}
```

Скелет кода для задания 3 (B)

Client_fifomsg.c

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"
#define MAXMESGDATA 4096

struct mymesg {
  long mesg_len;
  long mesg_type;
  char mesg_data[MAXMESGDATA];
        };
#define MESGHDRSIZE 2*sizeof(long)

void client(int readfd, int writefd);
ssize_t mesg_send(int fd, struct mymesg *mptr);
ssize_t mesg_recv(int fd, struct mymesg *mptr);

int main(int argc,char **argv)
{
        int readfd, writefd;

        if( (writefd=open(FIFO1, O_WRONLY))<0)
        {
          printf("Client: can not open FIFO1 for write\n");
                exit(1);
        }
  printf("Client: FIFO1 is opened for write writefd=%d\n",writefd);

        if( (readfd=open(FIFO2, O_RDONLY))<0)
        {
          printf("Client: can not open FIFO2 for read\n");
                exit(1);
        }
  printf("Client: FIFO2 is opened for read readfd=%d\n",readfd);

        client(readfd,writefd);

        close(readfd);
        close(writefd);

        if (unlink(FIFO1) < 0)
        {
           printf("Client: can delete FIFO1\n");
                    exit(1);
        }
```

```
  printf("Client: FIFO1 is deleted!\n");

        if (unlink(FIFO2) < 0)
        {
           printf("Client: can delete FIFO2\n");
                    exit(1);
        }
  printf("Client: FIFO2 is deleted!\n");

  printf("Client is terminated!\n");

        exit(0);
}

void client(int readfd, int writefd)
{
…
}

ssize_t mesg_send(int fd, struct mymesg *mptr)
{
  return(write(fd, mptr,MESGHDRSIZE + mptr->mesg_len));
}

ssize_t mesg_recv(int fd, struct mymesg *mptr)
{
  ssize_t n;
  size_t len;

  if (( n=read(fd,mptr,MESGHDRSIZE))==0) return(0); //end of file
  else if(n!=MESGHDRSIZE) { printf("Client: error MESGHDRSIZE\
n");return(0); }

  if((len=mptr->mesg_len)>0)
    if((n=read(fd,mptr->mesg_data,len))!=len)
    {
      printf("Client: message data expected len %d got n %d\n",len,n);
             exit(1);
    }
  return(len);
}
```

Server_fifomsg.c

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"
#define MAXMESGDATA 4096

struct mymesg {
  long mesg_len;
  long mesg_type;
  char mesg_data[MAXMESGDATA];
```

```
         };
#define MESGHDRSIZE 2*sizeof(long)

ssize_t mesg_send(int fd, struct mymesg *mptr);
ssize_t mesg_recv(int fd, struct mymesg *mptr);


void server(int readfd, int writefd);

int main(int argc,char **argv)
{
        int readfd, writefd;

        if( mknod(FIFO1, S_IFIFO | 0666, 0)<0)
        {
                printf("Server: can not create FIFO1\n");
                exit(1);
        }
  printf("Server: FIFO1 is created!\n");
        if( mknod(FIFO2, S_IFIFO | 0666, 0)<0)
        {
                unlink(FIFO1);
                printf("Server: can not create FIFO2\n");
                exit(1);
        }
  printf("Server: FIFO2 is created!\n");

        if( (readfd=open(FIFO1, O_RDONLY))<0)
        {
          printf("Server: can not open FIFO1 for read\n");
                exit(1);
        }
  printf("Server: FIFO1 is opened for read and readfd=%d\n",readfd);

        if( (writefd=open(FIFO2, O_WRONLY))<0)
        {
          printf("Server: can not open FIFO2 for write\n");
                exit(1);
        }
  printf("Server: FIFO2 is opened for write and writefd=%d\n",writefd);

        server(readfd,writefd);

        exit(0);
}

void server(int readfd, int writefd)
{
…
}

ssize_t mesg_send(int fd, struct mymesg *mptr)
{

  return(write(fd, mptr,MESGHDRSIZE+mptr->mesg_len));
}
```

```
ssize_t mesg_recv(int fd, struct mymesg *mptr)
{
  ssize_t n;
  size_t len;

  if((n=read(fd,mptr,MESGHDRSIZE))==0)
  {
    printf("Server: end of file\n");
          return(0);
  }
  else if(n!=MESGHDRSIZE) { printf("Server: error MESGHDRSIZE\n");
return(0); }
  //printf("Server: n=%d\n",mptr->mesg_len);

  if((len=mptr->mesg_len)>0)
    if((n=read(fd,mptr->mesg_data,len))!=len)
    {
      printf("Server: %s\n",mptr->mesg_data);
      printf("Server: can not read msg\n");
exit(1);
    }
  return(len);
}
```

Глава

4

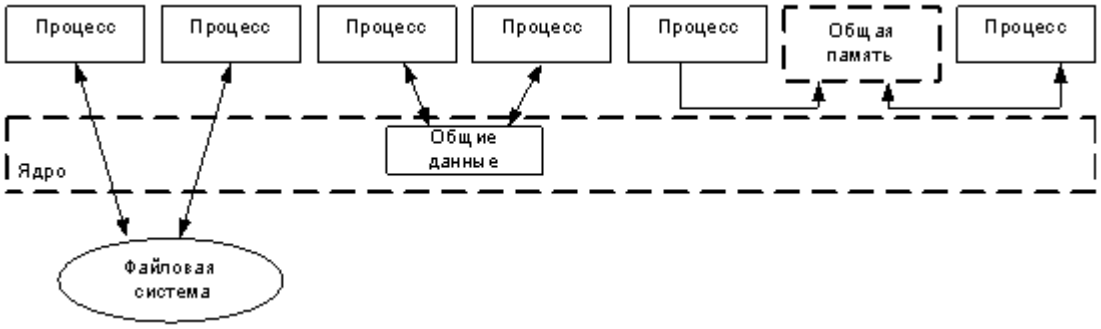
IPC

Введение

Совместное использование информации

ОС UNIX поддерживает одновременное выполнение нескольких процессов, каждому из которых выделяется собственное адресное пространство (*Рисунок 4.1*).

Рисунок 4.1 Совместное использование информации процессами



Два первых процесса (слева) совместно используют информацию, хранящуюся в файле. Для доступа к этим данным каждый процесс должен обратиться к ядру (используя функции **`read`**, **`write`**, **`lseek`**). Нужно использовать синхронизацию при изменении файла:

- чтобы исключить проблемы при одновременной записи в файл несколькими процессами;
- чтобы защитить процессы, читающие из файла, от процессов, которые записывают в файл.

Следующие два процесса (всередине) совместно используют информацию, хранящуюся в ядре. В качестве примеров можно привести очередь сообщений или семафор System V. Для доступа к совместно используемой информации в этом случае надо использовать системные вызовы.

Два процесса (справа) используют общую область памяти, к которой может обращаться каждый из процессов. После того как будет получен доступ к этой области памяти, процессы смогут обращаться к данным без помощи ядра, но при этом также необходима синхронизация.

Взаимодействующих процессов необязательно должно быть два, их может быть несколько.

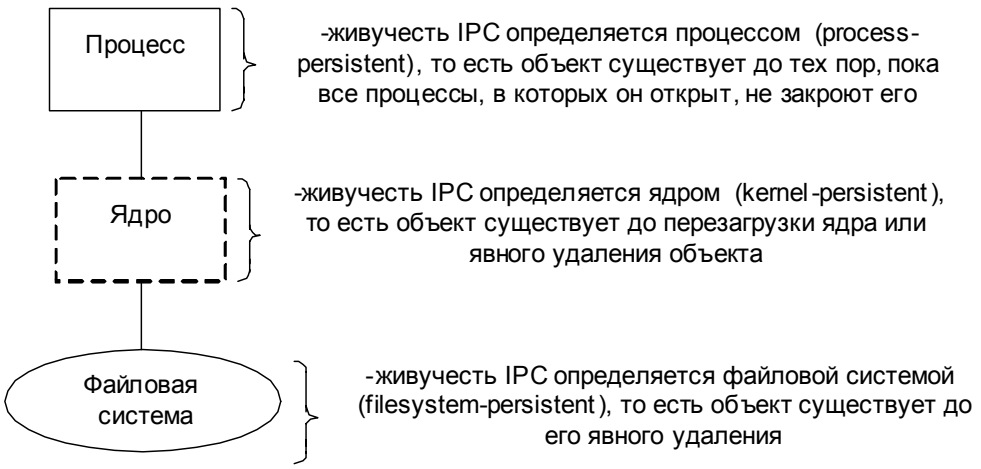
Нити (потоки)

Процессы в системах UNIX используется уже очень давно, а возможность использовать несколько нитей (потоков) внутри одного процесса появилась относительно недавно. Стандарт потоков Posix.1, называемый Pthreads, был принят в 1995 году. Все потоки одного процесса находятся в одном адресном пространстве и разделяют общие глобальные переменные. Однако при доступе к глобальным данным потокам требуется синхронизация для управления доступом к данным.

Живучесть объектов IPC

Можно определить живучесть (persistence) любого объекта IPC как продолжительность его существования. На рисунке (*Рисунок 4.2*) изображены три возможные группы, в которые входят объекты в соответствии со своей живучестью.

Рисунок 4.2 Живучесть объектов IPC



Живучесть различных объектов ICP (см. *Таблица 4.1*) :

- Объект IPC, ***живучесть*** которого ***определяется процессом*** (***process persistent***), существует до тех пор, пока не будет закрыт последним процессом, в котором он еще открыт. Это, например, неименованные и именованные каналы (pipes, FIFO).
- Объект IPC, ***живучесть*** которого ***определяется ядром*** (***kernel persistent***), существует до перезагрузки ядра или до тех пор, пока его не удалили. Это, например, очереди сообщений, семафоры и разделяемая память System V. Живучесть очередей сообщений, семафоров и разделяемой памяти Posix может определяться ядром, но может определяться и файловой системой (в зависимости от реализации).
- Объект IPC, ***живучесть*** которого ***определяется файловой системой*** (***filesystem persistent***), существует до тех пор, пока не будет удален. Такие объекты сохраняются даже при перезагрузке ядра. Это, например, очереди сообщений, семафоры и разделяемая память Posix, если они реализованы с использованием отображаемых в память файлов.

Таблица 4.1 Живучесть объектов IPC

Тип IPC	***Живучесть***
Программный канал (pipe)	Процесс
Именованный канал (FIFO)	Процесс
Очередь сообщений System V	Ядро
Семафор System V	Ядро
Разделяемая память System V	Ядро
Сокет TCP (TCP socket)	Процесс
Сокет UDP (UDP socket)	Процесс

Имена

Для того, чтобы два неродственных процесса могли использовать IPC для обмена информацией, объект IPC должен иметь имя или идентификатор:

- тогда один из процессов, обычно его называют ***сервер***, должен создать этот объект;
- а другой процесс (или несколько процессов), обычно его называют ***клиентом***, сможет обратиться к этому объекту.

Вспомним, что программные каналы (pipes) не имеют имен, поэтому их нельзя использовать для взаимодействия между неродственными процессами. Зато каналы FIFO имеют имена и поэтому их можно использовать для взаимодействия неродственных процессов.

В этой лекции рассматриваются другие типы IPC и используются дополнительные соглашения об именовании (naming conventions). Множество возможных имен для определенного типа IPC называется его ***пространством имен*** (***name space***). Пространство имен это важный термин, поскольку для всех видов IPC, за исключением простых каналов, именем определяется способ связи клиента и сервера для обмена сообщениями. В таблице (*Таблица 4.2*) приведены соглашения для различных типов IPC.

Таблица 4.2 Пространство имен IPC

Тип IPC	*Пространство имен*	*Идентификатор*
Канал	(Без имени)	Дескриптор
FIFO	Имя файла (pathname)	Дескриптор
Очередь сообщений System V	Ключ key_t	Идентификатор IPC System V
Семафор System V	Ключ key_t	Идентификатор IPC System V
Разделяемая память System V	Ключ key_t	Идентификатор IPC System V

System V IPC

Из имеющихся типов IPC к System V IPC относятся:

- ***очереди сообщений System V***;
- ***семафоры System V***;
- ***разделяемая память System V.***

Впервые они появились в UNIX System V, а были разработаны в конце 70-х годов одном из филиалов Bell Laboratories для одной из версий UNIX, которая предназначалась для внутреннего использования. Эта версия называлась Columbus Unix. System V IPC были добавлены в коммерческую версию UNIX System V в 1983 году.

Все три механизма похожи по программному интерфейсу и своей реализации, поэтому рассмотрим их общие черты. При описании их общих возможностей часто вместо термина объект IPC используется термин ***ресурс IPC*** (или просто ресурс). Каждый объект IPC обладает набором атрибутов:

- ***ключ*** (***key***) - поддерживаемое пользователем целое число, идентифицирующее конкретный экземпляр объекта;
- ***создатель*** (***creator***) - пользовательские и групповые идентификаторы процесса, создавшего объект;
- ***владелец*** (***owner***) - пользовательские и групповые идентификаторы владельца объекта. При создании объекта его владелец и создатель совпадают. Однако процесс, обладающий, правами изменения владельца, может позже стать владельцем нового объекта. Такими правами всегда обладает суперпользователь (root);
- ***права*** (***permissions***) - права файловой системы на чтение/запись/выполнение для владельца, группы и других пользователей.

Процесс получает доступ к соответствующему типу объекта при помощи системных вызовов:

- **`shmget`**;
- **`semget`**;
- **`msgget`**.

При этом он передает им ключ, необходимые флаги и другие аргументы, зависящие от используемого механизма. С помощью соответствующих флагов можно:

- Попросить ядро создать объект, если он не существует;
- Попросить ядро вернуть ошибку, если такой объект уже существует;
- Если флаги не указывать, то ядро будет искать существующий объект с тем же ключом:
  - Если такой объект существует и вызывающий процесс обладает соответствующими правами доступа (к нему), то ядро возвращает ***идентификатор ресурса*** (***ID***), который можно использовать в дальнейшем для быстрого доступа к объекту.

Для каждого механизма используются соответствующие управляющие системные вызовы:

- **`shmctl`**;

- **semctl**;
- **msgctl**.

С их помощью можно задать несколько различных команд:

- Для получения статусной информации (IPC_STAT);
- Для установки статусной информации (IPC_SET);
- Для освобождения объекта (IPC_RMID).

Необходимо учитывать, что статусная информация специфичной для каждого механизма. А для управления семафорами поддерживаются дополнительные команды управления, которые используются в целях получения и установки переменных отдельных семафоров набора.

Не забывайте, что каждый ресурс IPC надо освобождать принудительно. В противном случае ядро системы будет считать ресурс активным, даже если все использовавшие его процессы были завершены. Процесс может записать данные в разделяемую область памяти или в очередь сообщений и затем завершить свою работу. Эти данные может запросить позже другой процесс. Ресурс IPC является постоянным, т. е. годится для использования уже после завершения работы процесса, обращавшегося к нему.

Такой подход имеет недостатки, так как ядро системы не может определить, был ли ресурс оставлен активным для новых процессов или он был потерян случайно, например, если работа процесса была завершена принудительно, и тот не успел освободить ресурс.

Правами на удаление объекта обладают только процессы, являющиеся создателем, текущим владельцем объекта или обладающие привилегиями суперпользователя.

Информация о функциях приведена в таблице (*Таблица 4.3*).

Таблица 4.3 Функции System V IPC

Очереди сообщений	Семафоры	Общая память
Заголовочный файл <sys/msg.h>	<sys/sem.h>	<sys/shm.h>
Создание или открытие ***msgget***	***semget***	***shmget***
Операции управления ***msgctl***	***semctl***	***shmctl***
Операции IPC ***msgsnd msgrcv***	***semop***	***shmat shmdt***

Ключи и идентификаторы

Для таких объектов IPC, как очереди сообщений, семафоры и разделяемая память, процесс назначения имени является более сложным, чем просто указание имени файла. Имя для этих объектов называется ***ключом*** (***key***) и генерируется функцией **`ftok`** из двух компонентов - имени файла и идентификатора проекта:

```
key_t ftok(char *filename, char proj);
```

В качестве **`filename`** можно использовать имя некоторого файла, известное взаимодействующим процессам. Например, это может быть имя программы-сервера. Важно, чтобы этот файл существовал на момент создания ключа. Также нежелательно использовать имя файла, который создается и удаляется в процессе работы распределенного приложения, поскольку при генерации ключа используется номер ***индексного дескриптора файла*** (**`inode`**). Вновь созданный файл может иметь другой номер индексного дескриптора и впоследствии процесс, желающий иметь доступ к объекту, получит неверный ключ.

Пространство имен позволяет создавать и совместно использовать IPC неродственным процессам. Однако для ссылок на уже созданные объекты используются идентификаторы, точно так же, как файловый дескриптор используется для работы с файлом, открытым по имени.

Каждое из перечисленных IPC имеет свой уникальный дескриптор (идентификатор), используемый ОС (ядром) для работы с объектом. Уникальность дескриптора обеспечивается уникальностью дескриптора для каждого типа объектов (очереди сообщений, семафоры и разделяемая память), т.е. какая-либо очередь сообщений может иметь тот же численный идентификатор, что и разделяемая область памяти (хотя любые две очереди сообщений должны иметь различные идентификаторы).

Работа с объектами IPC System V похожа на работу с файлами в UNIX. Однако файловые дескрипторы имеют значение в контексте процесса, а дескрипторы объектов IPC распространяется на всю систему. Так файловый дескриптор 3 одного процесса в общем случае никак не связан с дескриптором 3 другого неродственного процесса (т.е. эти дескрипторы ссылаются на разные файлы). Иначе обстоит дело с дескрипторами объектов IPC. Все процессы, использующие, скажем, одну очередь сообщений, получат одинаковые дескрипторы этого объекта.

Разрешения

Для каждого из объектов IPC ядро поддерживает соответствующую структуру данных, отличную для каждого типа объекта IPC (очереди сообщений, семафора или разделяемой памяти). Общей у этих данных является структура ipc_perm, описывающая права доступа к объекту, подобно тому, как это делается для файлов. Основными полями этой структуры являются:

- **`UID`** - идентификатор владельца-пользователя объекта;
- **`GID`** - идентификатор владельца-группы объекта;
- **`CUID`** - идентификатор пользователя создателя объекта;
- **`CGID`** - идентификатор группы создателя объекта;
- **`mode`** - разрешения чтения/записи
- **`key`** - ключ объекта.

Права доступа (как и для файлов) определяют возможные операции, которые может выполнять над объектом конкретный процесс (получение доступа к существующему объекту, чтение, запись и удаление).

Заметим, что система не удаляет созданные объекты IPC даже тогда, когда ни один процесс ими не пользуется. Удаление созданных объектов является обязанностью процессов, которым для этого предоставляются соответствующие функции управления **`msgctl`**, **`semctl`**, **`shmctl`**. Напомним еще раз, что с помощью этих функций процесс может получить и установить ряд полей внутренних структур, поддерживаемых системой для объектов IPC, а также удалить созданные объекты. Безусловно, как и во многих других случаях использования объектов IPC процессы предварительно должны "договориться", какой процесс и когда удалит объект. Чаще всего, таким процессом является сервер.

Очереди сообщений

Как уже обсуждалось, очереди сообщений являются составной частью UNIX System V, они обслуживаются ОС, размещаются в адресом пространстве ядра и являются разделяемым системным ресурсом. Каждая очередь сообщений имеет свой уникальный идентификатор. Процессы могут записывать и считывать сообщения из различных очередей. Процесс, пославший сообщение в очередь, может не ожидать

чтения этого сообщения каким-либо другим процессом. Он может закончить свое выполнение, оставив в очереди сообщение, которое будет прочитано другим процессом позже.

Данная возможность позволяет процессам обмениваться структурированными данными, имеющими следующие атрибуты:

- Тип сообщения (позволяет мультиплексировать сообщения в одной очереди);
- Длина данных сообщения в байтах (может быть нулевой);
- Собственно данные (если длина ненулевая, могут быть структурированными).

Очередь сообщений хранится в виде внутреннего однонаправленного связанного списка в адресном пространстве ядра. Для каждой очереди ядро создает заголовок очереди (**msqid_ds**), где содержится информация о правах доступа к очереди (**msg_perm**), ее текущем состоянии (**msg_cbytes** - число байтов и **msg_qnum** - число сообщений в очереди), а также указатели на первое (**msg_first**) и последнее (**msg_last**) сообщения, хранящиеся в виде связанного списка (см. *Рисунок 4.3*). Каждый элемент этого списка является отдельным сообщением.

Для создания новой очереди сообщений или для доступа к существующей используется системный вызов **msgget**.

Функция возвращает дескриптор объекта-очереди (либо -1 в случае ошибки). Подобно файловому дескриптору, этот идентификатор используется процессом для работы с очередью сообщений. Процесс может:

- Помещать в очередь сообщения с помощью функции **msgsnd**;
- Получать сообщения определенного типа из очереди с помощью функции **msgrcv**;
- Управлять сообщениями с помощью функции **msgctl**.

Очереди сообщений обладают весьма полезным свойством - в одной очереди можно мультиплексировать сообщения от различных процессов. Для демультиплексирования используется атрибут **msgtype**, на основании которого любой процесс может фильтровать сообщения с помощью функции **msgrcv**.

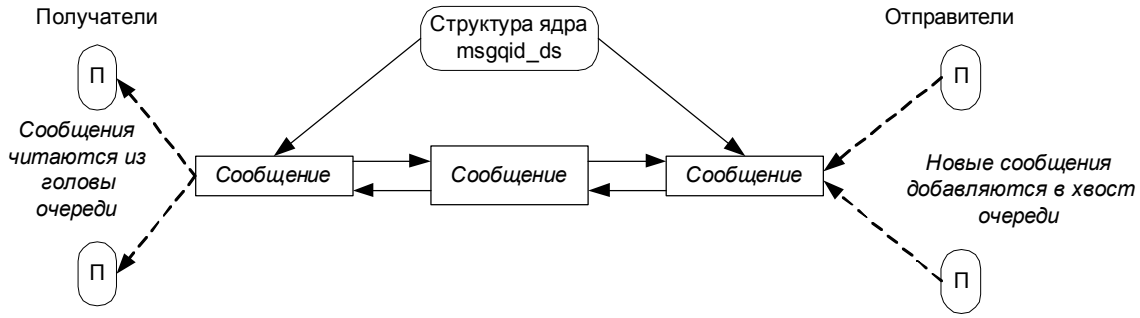
Рассмотрим взаимодействия процессов, когда серверный процесс обменивается данными с несколькими клиентами. Свойство мультиплексирования позволяет использовать для такого обмена одну

очередь сообщений. Для этого сообщениям, направляемым от любого из клиентов серверу, будем присваивать значение типа, скажем, равным 1. Если в теле сообщения клиент каким-либо образом идентифицирует себя (например, передает свой PID), то сервер сможет передать сообщение конкретному клиенту, присваивая тип сообщения равным этому идентификатору.

Поскольку функция **msgrcv** позволяет принимать сообщения определенного типа (типов), сервер будет принимать сообщения с типом 1, а клиенты - сообщения с типами, равными идентификаторам их процессов.

Атрибут msgtype также можно использовать для изменения порядка извлечения сообщений из очереди. Стандартный порядок получения сообщений аналогичен принципу FIFO - сообщения получаются в порядке их записи. Однако, используя тип, например, для назначения приоритета сообщений, этот порядок легко изменить.

Рисунок 4.3 Использование очереди сообщений



После прочтения сообщение удаляется из очереди и, следовательно, не может быть прочитано другими процессами. Процесс должен удалять очередь сообщений принудительно при помощи вызова **msgctl** с указанием команды IPC_RMID. При этом ядро освобождает очередь и удаляет все сообщения, находившиеся в ней.

ЛАБОРАТОРНАЯ РАБОТА 4

ОЧЕРЕДИ СООБЩЕНИЙ (IPC)

Теория

Ключи key_t и функция ftok

В таблице (см. *Таблица 4.2*) было отмечено, что в именах трех типов System V IPC использовались значения **key_t**. Заголовочный файл <sys/types.h> определяет тип **key_t** как целое (32-разрядное). Значения переменным этого типа обычно присваиваются функцией **ftok**.

Функция **ftok** преобразовывает существующее полное имя и целочисленный идентификатор в значение типа **key_t**, называемое ***ключом IPC*** (***IPC key***):

```
#include <sys/ipc.h>

key_t ftok(const char *pathname, int id);
//Возвращает ключ IPC либо -1 при возникновении ошибки
```

На самом деле функция использует полное имя файла, и младшие 8 бит идентификатора для формирования целочисленного ключа IPC.

Эта функция действует в предположении, что для конкретного приложения, использующего IPC, клиент и сервер используют одно и то же полное имя объекта IPC, имеющее какое-то значение в контексте приложения. Это может быть:

- имя файла данных, используемого сервером;
- или имя еще какого-нибудь объекта файловой системы.

Если клиенту или серверу для связи требуется только один канал IPC, идентификатору можно присвоить, например, значение 1. Если требуется несколько каналов IPC (например, один от сервера к клиенту и один в обратную сторону), идентификаторы должны иметь разные значения: например, 1 и 2. После того как клиент и сервер договорятся о полном имени и идентификаторе, они оба вызывают функцию **ftok** для получения одинакового ключа IPC.

Если указанное полное имя не существует или недоступно вызывающему процессу, **ftok** возвращает значение -1.

Помните, что файл, имя которого используется для вычисления ключа, не должен создаваться и удаляться сервером в процессе работы, поскольку каждый раз при создании заново файл получает другой номер индексного дескриптора. И мы можем получить другой ключ, возвращаемый функцией **ftok** при очередном вызове.

Структура ipc_perm

Для каждого объекта IPC, как для обычного файла, в ядре хранится структура:

```
struct ipc_perm {
   uid_t uid; // идентификатор пользователя владельца
   gid_t gid; // идентификатор группы владельца
   uid_t cuid; // идентификатор пользователя создателя
   gid_t cgid; // идентификатор группы создателя
   mode_t mode; // разрешения чтения/записи
   ulong_t seq; // последовательный номер канала
   key_t key; // ключ IPC
     };
```

Эта структура вместе с другими переименованными константами для функций System V IPC определена в файле <sys/ipc.h>. Далее поля этой структуры будут рассмотрены более подробно.

Создание и открытие System V IPC

Три функции **get<>**, используемые для создания или открытия объектов IPC (см. *Таблица 4.3*), принимают ключ **key_t** в качестве одного из аргументов и возвращает целочисленный идентификатор.

У приложения есть две возможности задания ключа первого аргумента функций **get<>**:

- Вызвать **ftok** и передать ей полное имя и идентификатор;
- Указать в качестве ключа константу **IPC_PRIVATE**, гарантирующую создание нового уникального объекта IPC.

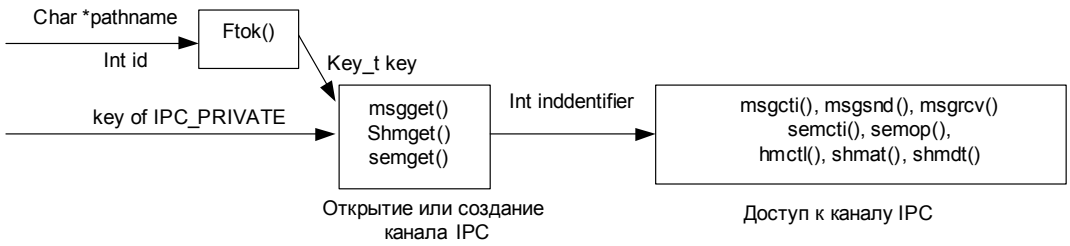
Последовательность действий приведена на рисунке (*Рисунок 4.4*).

Все три функции **get<>** принимают в качестве второго аргумента набор флагов **oflag**, задающий биты разрешений чтения/записи (поле **mode** структуры **ipc_perm**) для объекта IPC и определяющий, создается ли новый объект IPC или производится обращение к уже существующему объекту. Используются следующие правила:

- Ключ **IPC_PRIVATE** гарантирует создание уникального объекта IPC. Никакие возможные комбинации полного имени и идентификатора не могут привести к тому, что функция **ftok** вернет в качестве ключа значение **IPC_PRIVATE**.

- Установка бита **IPC_CREAT** аргумента **oflag** приводит к созданию новой записи для указанного ключа, если она еще не существует. Если же обнаруживается существующая запись, возвращается ее идентификатор.

Рисунок 4.4 Определение идентификаторов IPC по ключам



Одновременная установка битов **IPC_CREAT** и **IPC_EXCL** и аргумента oflag приводит к созданию новой записи для указанного ключа только в том случае, если такая запись еще не существует. Если же обнаруживается существующая запись, функция возвращает ошибку **EEXIST** (объект IPC уже существует).

Логическая диаграмма последовательности действий при открытии объекта IPC изображена рисунке (*Рисунок 4.5*).

При создании нового объекта IPC с помощью одной из функций **get<>**, вызванной с флагом **IPC_CREAT**, в структуру **ipc_perm** заносится следующая информация:

- часть битов аргумента **oflag** задают значение поля **mode** структуры **ipc_perm**. В таблице (*Таблица 4.4*) приведены биты разрешений для трех типов IPC (запись >> 3 означает сдвиг вправо на три бита);

- поля **cuid** и **cgid** получают значения, равные действующим идентификаторам пользователя и группы вызывающего процесса. Эти два поля называются идентификаторами создателя;

- поля **uid** и **gid** структуры **ipc_perm** также устанавливаются равными действующим идентификаторам вызывающего процесса. Эти два поля называются идентификаторами владельца.

Рисунок 4.5 Открытие объекта IPC

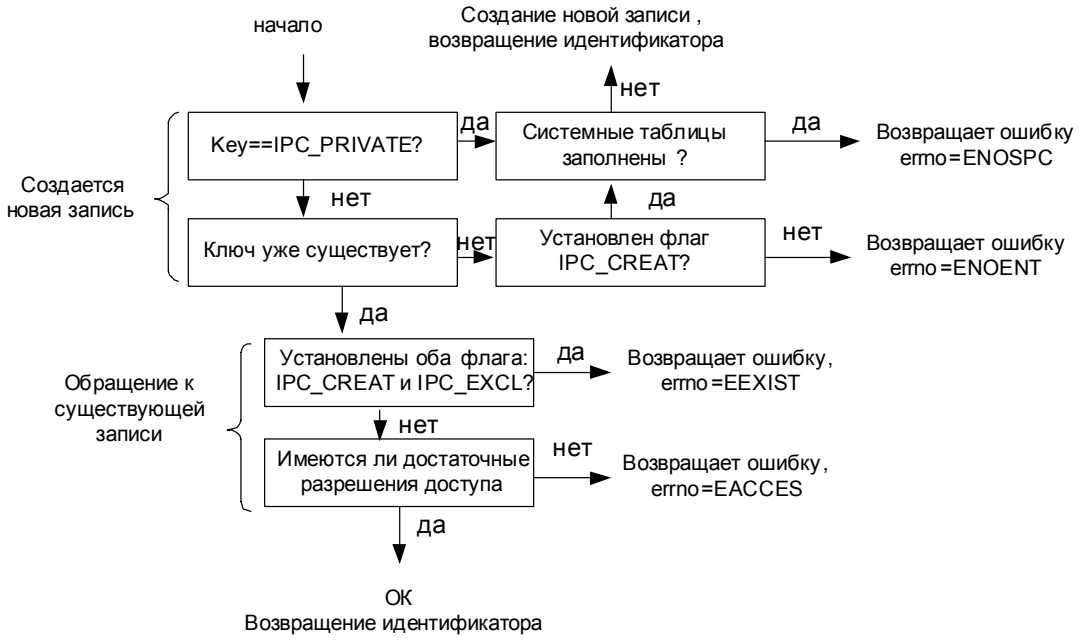


Таблица 4.4 Значения **mode** для разрешений чтения/запись

Число (восьмеричное)	Очередь сообщений	Семафор	Разделяемая память	Описание
0400	MSG_R	SEM_R	SHM_R	Пользователь-чтение
0200	MSG_W	SEM_A	SHM_W	Пользователь-запись
0040	MSG_R>>3	SEM_R>>3	SHM_R>>3	Группа-чтение
0020	MSG_W>>3	SEM_A>>3	SHM_W>>3	Группа-запись
0004	MSG_R>>6	SEM_R>>6	SHM_R>>6	Прочие-чтение
0002	MSG_W>>6	SEM_A>>6	SHM_W>>6	Прочие-запись

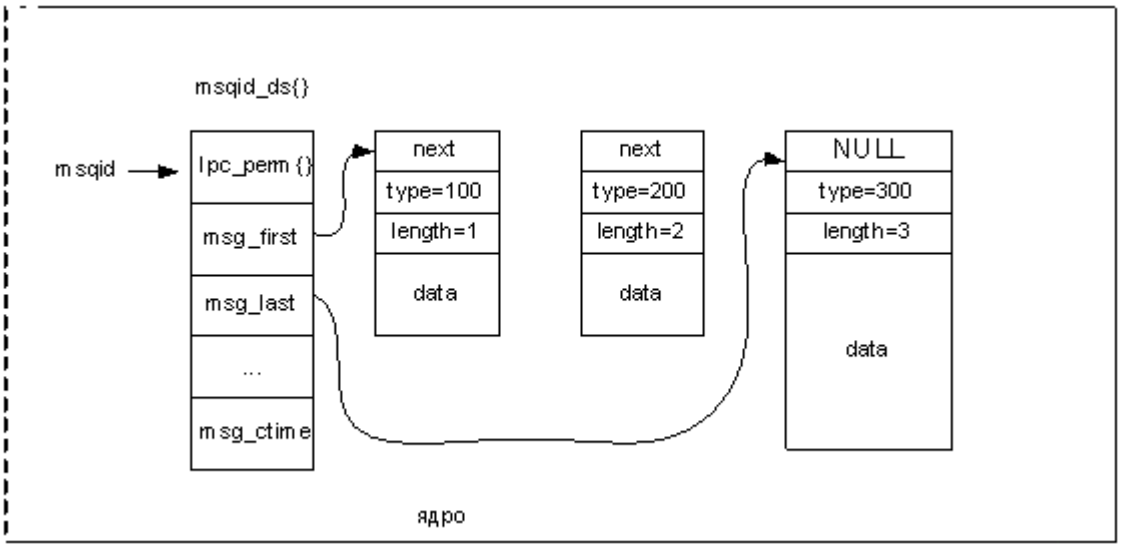
Очереди сообщений

Очередь сообщений - это заголовок, указывающий на связанный список сообщений (см. *Рисунок 4.6*).

Каждое сообщение содержит 32-разрядную переменную типа, следующую за областью данных. Процесс создает или получает очередь сообщений при помощи системного вызова **msgget**:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

msgpid = msgget(key, flag);
```

Рисунок 4.6 Структура очереди сообщений System V



Здесь **key** - это целое число, задаваемое пользователем. Для создания новой очереди сообщений необходимо указать флаг **IPC_CREAT**. Задание флага **IPC_EXCL** ведет к ошибочному завершению работы вызова в том случае, если очередь с указываемым ключом уже существует. Переменная **msgpid** используется в дальнейших вызовах для доступа к очереди.

Для того чтобы поместить сообщение в очередь, необходимо использовать:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

msgsnd(msgqid, msgp, count, flag);
```

Здесь **msqid** является дескриптором объекта, полученного в результате вызова **msgget**. Параметр **msgp** указывает на буфер, содержащий тип сообщения и его данные, размер которого равен **msgsz** байт. Буфер имеет следующие поля:

- **long msgtype** - тип сообщения
- **char msgtext[]** - данные сообщения

Если **msgtyp** равен 0, функция **msgrcv** получит первое сообщение из очереди. Если величина **msgtype** больше 0, будет получено первое сообщение указанного типа. Если меньше 0, функция **msgrcv** получит сообщение с минимальным значением типа, меньше или равного абсолютному значению **msgtype**.

Флаг **IPC_NOWAIT** используется для возврата ошибки, если сообщение невозможно отправить без блокировки(например, когда очередь переполнена, так очередь обычно обладает настраиваемым ограничением на количество хранящихся в ней данных).

На рисунке (*Рисунок 4.3*) показаны операции над очередями сообщений. Каждая очередь описывается в виде строки в таблице ресурсов очередей сообщений.

```
struct msqid_ds {
struct ipc_perm msg_perm; /* разрешения */
struct msg* msg_first; /*указатель на первое сообщение в очереди*/
struct msg*msg_last; /*указатель на последнее сообщение в очереди*/
ushort msg_cbytes; /*размер очереди в байтах*/
ushort msg_qnum; /*количество сообщений в очереди*/
ushort msg_qbytes; /*максимально допустимый размер очереди в байтах*/
…
};
```

Сообщения располагаются внутри очереди в порядке их поступления. Они удаляются из очереди по принципу "первым вошел, первым вышел" при чтении их процессом при помощи вызова:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

count=msgrcv(msgqid, msgp, maxcnt, msgtype, flag);
```

msgp указывает на буфер, в который помещается входящее сообщение, maxcnt ограничивает максимально прочитываемое количество байтов. Если входящее сообщение длиннее, чем **maxcnt**, то оно будет обрезано. Пользователь должен быть уверен в том, что буфер, указанный при помощи **msgp**, имеет достаточный объем для хранения maxcnt байтов данных. Возвращаемая функцией величина указывает на успешно прочитанное количество байтов.

Чему нужно научиться?

Изучить использование очередей сообщений System V.

Задание А

Уровень 1 (А)

Посмотрите на Скелет кода для задания 1 (А) и **разберитесь, что там происходит**.

Уровень 2 (А)

Напишите программы для сервера и клиента. Одновременно могут работать несколько клиентов.

Сервер должен включать обработчик сигнала **SIGINT** (с восстановлением диспозиции и удалением очереди сообщений системным вызовом **msgctl()** для корректного завершения сервера при получении сигнала **SIGINT**). Создайте очередь сообщений, используя системный вызов **msgget(key, PERM | IPC_CREAT)**.

Сервер в цикле ожидает сообщение, а затем читает его из очереди (тип 1) и посылает на каждое сообщение ответ клиенту (тип 2). Выводите сообщения на экран для проверки. Если происходит ошибка, используете функцию **kill()** (отправьте сигнал **SIGINT**).

Client.c должен получить доступ к очереди сообщений, а затем отправить сообщение серверу (тип 1). Клиент ожидает сообщение, а затем читает его (тип 2). Выведите сообщение на экран для проверки.

Для чтения и посылки сообщения используйте системные вызовы **msgrcv()** и **msgsnd()**.

Откомпилируйте программы (**gcc -o server server.c; gcc -o client client.c**), запустите на выполнение исполняемые модули **./server** и **./client**, отправьте процессу server сигнал **SIGINT**.

Создайте несколько клиентов, запустите их одновременно.

Уровень 3 (А)

Посмотрите на Скелет кода для задания 3 (А), **разберитесь, что там происходит, и допишите недописанные функции**.

Скелет кода для задания 1 (А)

mesg.h

```
#define MAXBUFF 80
#define PERM 0666
/*Определим структуру сообщения*/
typedef struct our_msgbuf {
  long mtype;
  char buff[MAXBUFF];
  } Message;
```

Server.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

main()
{
Message message;
key_t key;
int msgid, length, n;

/*Получим ключ*/
if((key = ftok("server", 'A'))<0)
{ printf("Невозможно получить ключ\n"); exit(1); }

/*Тип принимаемых сообщений*/
message.mtype = 1L;

  /*Создадим очередь сообщений*/
  if((msgid = msgget(key,PERM|IPC_CREAT))<0)
  { printf("Невозможно создать очередь\n"); exit(1); }

  /*Примем сообщение*/
  n = msgrcv(msgid, &message, sizeof(message), message.mtype, 0);

  /*Если сообщение поступило, выведем его содержимое на терминал*/
  if(n>0){
    if(write(1, message.buff,n) != n){ printf("Ошибка вывода\n"); exit(1);}
    }
  else { printf("Ошибка чтения сообщения\n"); exit(1); }
  /*Удалить очередь должен клиент*/
  exit(0);
}
```

Client.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include "mesg.h"

main()
{

  Message message;
  key_t key;
  int msgid, length;

  /*Тип посылаемого сообщения*/
  message.mtype = 1L;

  /*Получим ключ*/
  if((key = ftok("server", 'A'))<0)
  { printf("Невозможно получить ключ\n"); exit(1); }
```

```
  /*Получим доступ к очереди сообщений*/
  /*очередь уже должна быть создана сервером*/
  if((msgid = msgget(key, 0))<0)
  { printf("Невозможно получить доступ кочереди\n"); exit(1); }

  /*Подготоим сообщение*/
  if ((length = sprintf(message.buff,"IPC Messages!\n"))<0)
  { printf("Ошибка копирования в буфер\n"); exit(1); }

  /*Передадим сообщение*/
  if(msgsnd(msgid, (void*) &message, length, 0) != 0)
  { printf("Ошибка записи сообщения в очередь\n); exit(1); }

  /*Удалим очередь сообщений*/
  if(msgctl(msgid, IPC_RMID, 0)<0)
  { printf("Ошибка удаления очереди\n"); exit(1);}
  exit(0);
}
```

Скелет кода для задания 3 (А)

client_msg.c

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/msg.h>
#define MQ_KEY1 1234L
#define MQ_KEY2 2345L
#define MAXMESGDATA 4096

struct mymesg {
  long mesg_len;
  long mesg_type;
  char mesg_data[MAXMESGDATA];
  };

void client(int readid, int writeid);
ssize_t mesg_send(int id, struct mymesg *mptr);
ssize_t mesg_recv(int id, struct mymesg *mptr);

int main(int argc,char **argv)
{
  int readid, writeid;

  if( (writeid=msgget(MQ_KEY1,0666)) <0)
  {
    printf("Client: can not get writeid!\n"); exit(1);
  }
  printf("Client:writeid=%d\n",writeid);

  if((readid=msgget(MQ_KEY2,0666)) <0)
  {
    printf("Client: can not get readid!\n"); exit(1);
```

```
  }
  printf("Client: readid=%d\n",readid);

  client(readid,writeid);

  if((msgctl(readid,IPC_RMID, NULL)) < 0)
  {
    printf("Client: can not delete massage queue2!\n"); exit(1);
  }

  if((msgctl(writeid,IPC_RMID, NULL)) <0)
  { printf("Client: can not delete massage queue1!\n"); exit(1); }

  exit(0);
}

void client(int readid, int writeid)
{
  size_t len;
  ssize_t n;
  struct mymesg ourmesg;

  printf("Client:readid=%d writeid=%d\n",readid,writeid);

  fgets(ourmesg.mesg_data,MAXMESGDATA, stdin);
  len=strlen(ourmesg.mesg_data);

  if(ourmesg.mesg_data[len-1]=='\n') len--;
  ourmesg.mesg_len=len;

  ourmesg.mesg_type=1;

  printf("Client: %s\n",ourmesg.mesg_data);

  mesg_send(writeid,&ourmesg);

  printf("Client: before recv!\n");

  while((n= mesg_recv(readid, &ourmesg))>0)
    write(1,ourmesg.mesg_data, n);
}

ssize_t mesg_send(int id, struct mymesg *mptr)
{
  …
}

ssize_t mesg_recv(int id, struct mymesg *mptr)
{
  …
}
```

Server_msg.c

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/msg.h>

#define MQ_KEY1 1234L
#define MQ_KEY2 2345L
#define MAXMESGDATA 4096

struct mymesg {
  long mesg_len;
  long mesg_type;
  char mesg_data[MAXMESGDATA];
  };

void server(int readid, int writeid);
ssize_t mesg_send(int id, struct mymesg *mptr);
ssize_t mesg_recv(int id, struct mymesg *mptr);

int main(int argc,char **argv)
{
  int readid, writeid;
  key_t key1, key2;

  printf("Server: HELLO!\n");

/*
  if((key1=ftok("/home/tvk/IPC/input.txt",'A'))<0)
  {
    printf("Server: can not get key!\n"); exit(1);
  }
  printf("key1=%x\n",key1);
*/
  if((readid=msgget(MQ_KEY1, 0666|IPC_CREAT))<0)
  {
    printf("Server: can not get readid!\n"); exit(1);
  }
  printf("Server: readid=%d\n",readid);
/*
  if((key2=ftok("/home/tvk/IPC/server_msg.c",'B'))<0)
  {
    printf("Server: can not get key!\n"); exit(1);
  }
  printf("key2=%x\n",key2);
*/
  if((writeid=msgget(MQ_KEY2, 0666|IPC_CREAT))<0)
  {
    printf("Server: can not get readid!\n"); exit(1);
  }
  printf("Server: writeid=%d\n",writeid);

  server(readid,writeid);
```

```
  exit(0);
}

void server(int readid, int writeid)
{
  FILE *fp;
  ssize_t n;
  struct mymesg ourmesg;

  printf("Server:readid=%d writeid=%d\n",readid,writeid);

  ourmesg.mesg_type=1;

  if( (n=mesg_recv(readid, &ourmesg)) == 0)
  { printf("Server: can not read file name\n"); exit(1); }
  ourmesg.mesg_data[n]='\0';

  printf("Server: file name %s\n",ourmesg.mesg_data);

  if( (fp=fopen(ourmesg.mesg_data,"r"))==NULL)
  { printf("Server: can not open file name\n"); }
  else
  {
    printf("Server: %s is opened\n",ourmesg.mesg_data);

  while(fgets(ourmesg.mesg_data, MAXMESGDATA,fp) != NULL)
    {
      ourmesg.mesg_len=strlen(ourmesg.mesg_data);
      printf("Server: %s\n",ourmesg.mesg_data);
      mesg_send(writeid,&ourmesg);
    }
  }
  fclose(fp);
  ourmesg.mesg_len=0;
  mesg_send(writeid,&ourmesg);
}

ssize_t mesg_send(int id, struct mymesg *mptr)
{
  …
}

ssize_t mesg_recv(int id, struct mymesg *mptr)
{
  …
}
```

Глава

5

Синхронизация

Введение

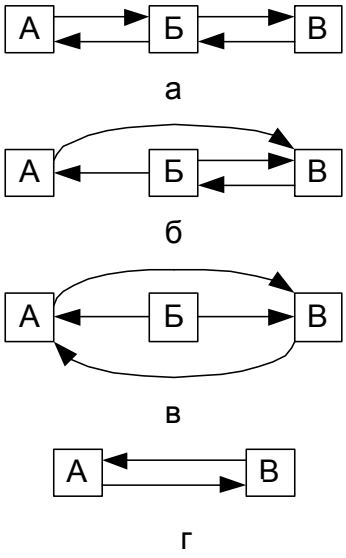
Рассмотрим еще раз, зачем нужен механизм синхронизации и что может случиться, если его не использовать. Пусть процесс пытается удалить элемент Б из связанного списка. Рассмотрим, как это происходит по шагам:

- сначала удаляем связь АБ и устанавливаем связь АВ (б);
- затем удаляем связь ВБ и устанавливаем связь ВА (в)

Если после выполнения первого шага на процессор будет назначен другой процесс, и он попытается получить доступ к тому же списку, то обнаружит его в противоречивом состоянии (см. *Рисунок 5.1*). Необходимо использовать некий механизм, защищающий от возникновения подобных проблем.

UNIX это многозадачная система, поэтому в любой момент времени в состоянии "***готов к выполнению***" (***ready***) могут находиться сразу несколько процессов. На однопроцессорных системах только один из них будет назначен на выполнение, а остальные будут ожидать освобождения процессора. Так, как все эти процессы используют одни и те же структуры данных ядра, необходима синхронизация, для поддержки этих структур в непротиворечивом состоянии.

Рисунок 5.1 Удаление элемента из связанного списка



В системе UNIX применяется несколько различных технологий синхронизации. Традиционное ядро UNIX было разработано ***невытесняющим***. Это означает, что процесс (нить), выполняющийся в режиме ядра, не может быть вытеснен другим процессом (нитью), даже если отведенный ему квант времени уже истек. Приводимые далее рассуждения справедливы не только для процессов, но и для нитей (вместо термина процесс можно использовать термин нить). Процесс должен самостоятельно процессор. Это обычно происходит в тот момент когда:

- процесс приостанавливает свою работу в ожидании необходимого ресурса или какого-то события;
- процесс завершил функционирование в режиме ядра и собирается возвращаться в режим задачи.

В любом случае из этих случаев, процесс освобождает процессор добровольно, поэтому он может быть уверен в том, что структуры ядра системы находится в корректном (непротиворечивом) состоянии.

Операции блокировки

В UNIX защита ресурсов реализована при помощи флагов ***locked*** (***ресурс занят***) и ***wanted*** (ресурс необходим). Если нити необходимо получить доступ к разделяемому ресурсу (например, буферу), то в первую очередь ей нужно проверить состояние флага **`locked`**. Если флаг не установлен, нить установит его и начнет работать с ресурсом. Если

другая нить попытается получить доступ к тому же ресурсу, она обнаружит флаг **`locked`** и приостановит работу ("заснет") до тех пор, пока ресурс не станет доступным. Перед блокировкой нить установит для ресурса флаг **`wanted`**. Приостановка работы нити означает помещение ее в очередь спящих нитей и изменение ее состояния. Когда нить закончит работать с ресурсом, сбросит флаг **`locked`** и проверит флаг **`wanted`**. Если флаг окажется установленным, то это значит, что, по крайней мере, еще одна нить ждет освобождения этого ресурса. В этом случае ОС проверит очередь спящих нитей и разбудит все нити, которые хотели этого ресурса. Что ОС делает, чтобы разбудить нить:

- удаляет ее из очереди спящих нитей;
- изменяет ее состояние на готова к выполнению (ready);
- и помещение ее в очередь планировщика.

В какой-то момент времени нить будет назначена на процессор - ***состояние выполнения*** (***running***). Первая ее задача при получении процессорного времени - проверка флага **`locked`**. Если флаг не установлен, то нить может использовать необходимый ей ресурс.

Операция блокировки - это операция, которая блокирует процесс (переводит его в спящее состояние до тех пор, пока блокировка не будет снята).

Для защиты объектов ядро ассоциирует с ними флаг, значение которого устанавливается, если объект заблокирован и сбрасывается в противном случае. Перед тем как начать пользоваться каким-либо объектом, каждый процесс должен проверять, не заблокирован ли требуемый объект. Если тот свободен, процесс устанавливает флаг блокировки, и начинает использовать ресурс. На рисунке (*Рисунок 5.2*) показан алгоритм блокировки ресурсов.

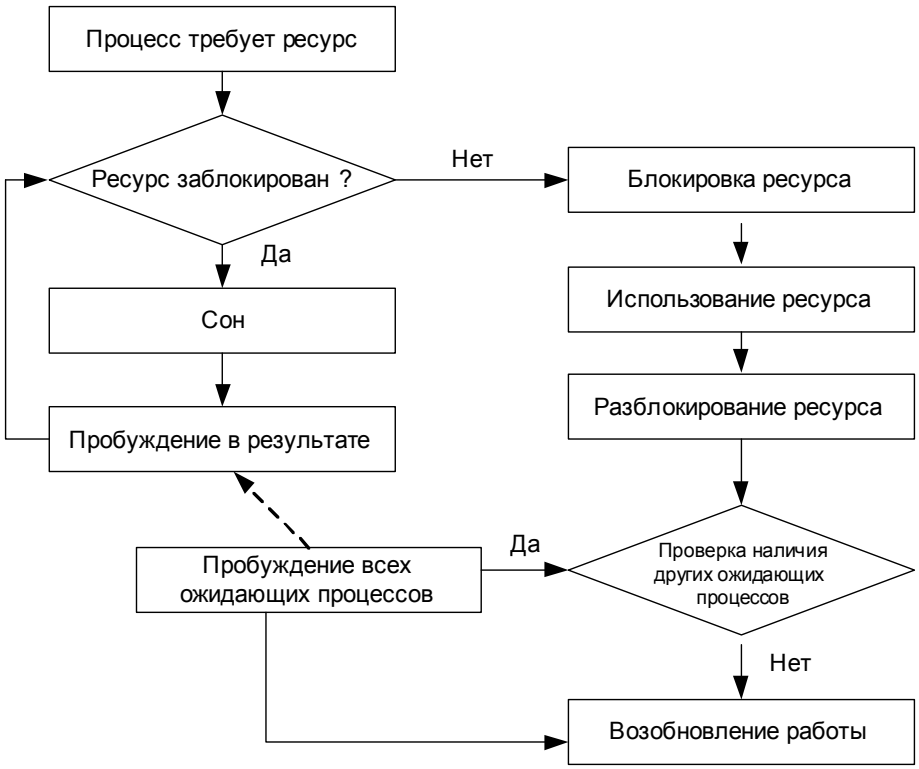
Необходимо учитывать следующее:

- процесс блокируется в том случае, если не может получить необходимый ресурс, или в случае ожидания наступления определенного события (например, завершения ввода/вывода). Для этого процесс вызывает процедуру **`sleep()`**. Это называется блокировкой по событию или ресурсу;
- процедура **`sleep()`**:
  - помещает процесс в очередь блокированных процессов;
  - изменяет его состояние на сон;

  - и вызывает функцию **`swtch()`** для инициализации переключения контекста и разрешения выполнения следующего процесса;
- процесс, освобождающий ресурс, вызывает процедуру **`wakeup()`** для пробуждения всех процессов, ожидающих этот ресурс. Функция **`wakeup()`**:
  - находит все ожидающие этот ресурс процессы;
  - изменяет их состояние на готов к выполнению (**`ready`**);
  - и помещает их в очередь планировщика, в которой они ждут, когда их назначат на выполнение.

Промежуток времени, который проходит между тем моментом, когда процесс разбудили, и временем, когда подошла его очередь выполняться, может быть большим. Возможна ситуация, когда другие процессы снова займут необходимый нашему процессу ресурс, поэтому он должен снова проверить, доступен ли ресурс. Если ресурс успели занять, то процесс снова переходит в спящий режим.

Рисунок 5.2 Блокировка ресурсов



Прерывания

Несмотря на то, что ядро системы защищено от вытеснения другими процессами, процесс, манипулирующий структурами данных ядра, может быть прерван различными устройствами. Если обработчик прерывания попытается получить доступ к таким структурам, то обнаружит, что они находятся в состоянии нарушения целостности.

Еще раз повторим суть этого механизма при использовании нитей. Текущую выполняемую нить невозможно вытеснить, но ее работу можно прервать. Прерывания являются внутренними событиями системы и должны обрабатываться как можно быстрее. Обработчик прерывания тоже может начать манипулировать структурами данных ядра, с которыми не закончила работу прерванная нить. Это может привести к нарушению целостности данных. Следовательно, нужно синхронизовать доступ к структурам данных ядра, которые используются программами и обработчиками прерываний.

В системе UNIX эта проблема решается при помощи блокировки (или маскирования) прерываний. Каждому прерыванию присваивается ***уровень приоритета*** (***interrupt priority level***, ***ipl***). Система поддерживает текущий уровень ***ipl*** и проверяет его после возникновения прерывания. Если приоритет возникшего прерывания окажется выше текущего значения, то такое прерывание будет обработано немедленно (то есть будет приостановлена обработка прерывания с боле низким уровнем ***ipl***). В противоположном случае ядро заблокирует прерывание до тех пор, пока текущий уровень ***ipl*** не снизится до необходимого уровня. Сразу перед загрузкой обработчика ядро системы устанавливает ***ipl*** в значение текущего прерывания, после завершения его обработки ядро восстанавливает предыдущее сохраненное значение. Ядро может установить значение ***ipl*** в принудительном порядке для временной блокировки прерываний в течение выполнения критических участков кода.

Таким образом, можно блокировать прерывания при доступе к критически важным структурам данных, но в этом случае нужно учитывать следующие важные обстоятельства:

- прерывания обычно нужно обработать как можно быстрее, поэтому их нельзя запрещать надолго. Таким образом, критические области кода должны быть по возможности короткими и их должно быть мало;
- необходимо блокировать только те прерывания, обработка которых требует обращения к данным, использующимся в критической области;

- два различных прерывания могут иметь один и тот же уровень приоритета;
- блокирование прерывания приводит к блокированию всех прерываний, имеющих такой же или более низкий уровень приоритета.

Синхронизация в многопроцессорных системах

Повторим еще раз, какие проблемы решаются за счет ***невытесняемости*** ядра. Процесс, выполняющийся в режиме ядра, не может быть вытеснен другим процессом, даже если отведенный ему квант времени уже истек. Процесс освобождает процессор добровольно и поэтому может быть уверен в том, что структуры данных ядра находится в корректном (непротиворечивом) состоянии. В однопроцессорных системах нужно защищать только те структуры данных, до которых могут добраться обработчики прерываний, а также те, целостность которых зависит от работы вызова **`sleep()`**.

С появление многопроцессорных систем возникли новые проблемы, связанные с синхронизацией. В многопроцессорных системах два процесса могут:

- одновременно выполняться в режиме ядра на разных процессорах;
- а также выполнять параллельно одну и ту же функцию.

Таким образом, каждый раз, когда ядро обращается к глобальным структурам данных, теперь оно должно защищать их так, чтобы к ним не смогли получить доступ процессы, выполняющиеся на других процессорах. Сами механизмы защиты также должны быть защищены от особенностей выполнения в многопроцессорных системах. Если два процесса, выполняющиеся на различных процессорах, попытаются одновременно заблокировать один и тот же объект, только один должен завершить успешно эту процедуру.

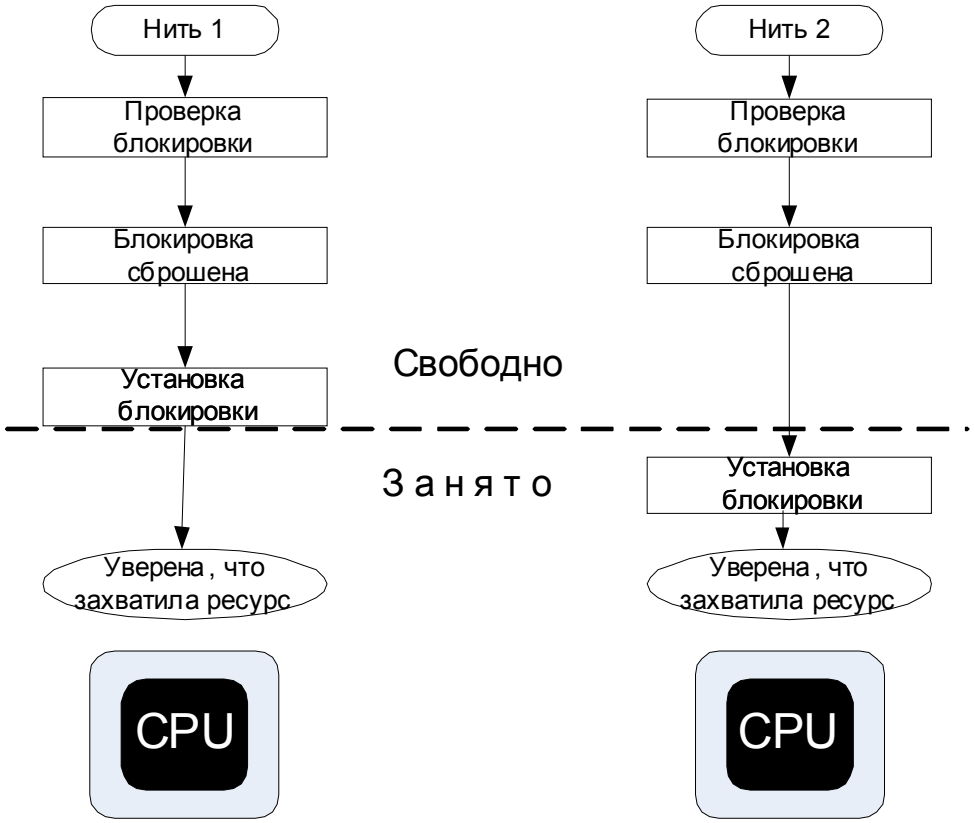
Неделимая команда тестирования и установки

Синхронизация в многопроцессорных системах сильно зависит от ее аппаратной поддержки. Рассмотрим, как проходит блокировка ресурса при помощи флага **`locked`** в области разделяемой памяти:

- чтение флага;
- если значение равно нулю (следовательно, ресурс свободен), то блокируем ресурс, устанавливая флаг в единицу;

- возвращаем **TRUE** в случае удачной блокировки ресурса или **FALSE** в противоположном случае.

Рисунок 5.3 Неделимость операции тестирования и установки



В многопроцессорной системе две нити, выполняющиеся на двух различных процессорах, могут производить эти операции одновременно. При этом обе нити будут считать, что обладают эксклюзивным правом на ресурс (см. *Рисунок 5.3*). В традиционных системах ядро просто проверяет флаг locked и устанавливает его для блокировки объекта. В многопроцессорных архитектурах может случиться так, что один и тот же флаг locked начнет проверяться двумя нитями одновременно. В этом случае обе нити обнаружат флаг сброшенным и сделают вывод о том, что ресурс свободен. Затем нити установят флаг и начнут использовать объект, что может привести к совершенно неожиданным результатам. Следовательно, многопроцессорные системы должны поддерживать неделимые операции проверки и установки флагов, гарантирующие занятие ресурса одновременно только одной нитью.

Для предупреждения возникновения подобных ситуаций аппаратура компьютера должна предлагать конструкцию, объединяющую все три операции. Существует реализация этой задачи, основанная на применении инструкции тестирования и установки.

Неделимая операция проверки и установки (***test-and-set***) обычно используется для действий над одним битом. Команда проверяет его значение, устанавливает в единицу и возвращает предыдущее значение. Таким образом, после завершения ее работы бит равняется 1 (ресурс занят), а возвращаемая величина показывает предыдущее значение этого бита. Если две нити, выполняющиеся на двух различных процессорах, попытаются одновременно произвести действия над одним и тем же битом, то свойство неделимости операции даст им возможность сделать это только последовательно, друг за другом. Это действие также неделимо с точки зрения прерываний. Если прерывание возникнет во время выполнения команды, то будет обработано только после ее завершения.

Операция тестирования и установки идеально подходит для простых случаев блокировки ресурсов. Если команда возвращает единицу, вызвавшая ее нить занимает ресурс. Если возвращает ноль, то требуемый ресурс уже занят другой нитью. Освобождение ресурса производится путем сброса бита в 0.

Одним из основных условий, на которых построена традиционная модель синхронизации, является непрерываемое использование структур данных ядра нитью до тех пор, пока она не выйдет из режима ядра самостоятельно или не будет заблокирована в ожидании ресурса (кроме случаев возникновения прерываний). Такое свойство однопроцессорных ОС совершенно не подходит для многопроцессорных систем, в которых нити ядра могут выполняться одновременно на нескольких CPU. Данные, не требовавшие защиты от повреждения при использовании единственного процессора, теперь необходимо защищать.

Обработка прерываний

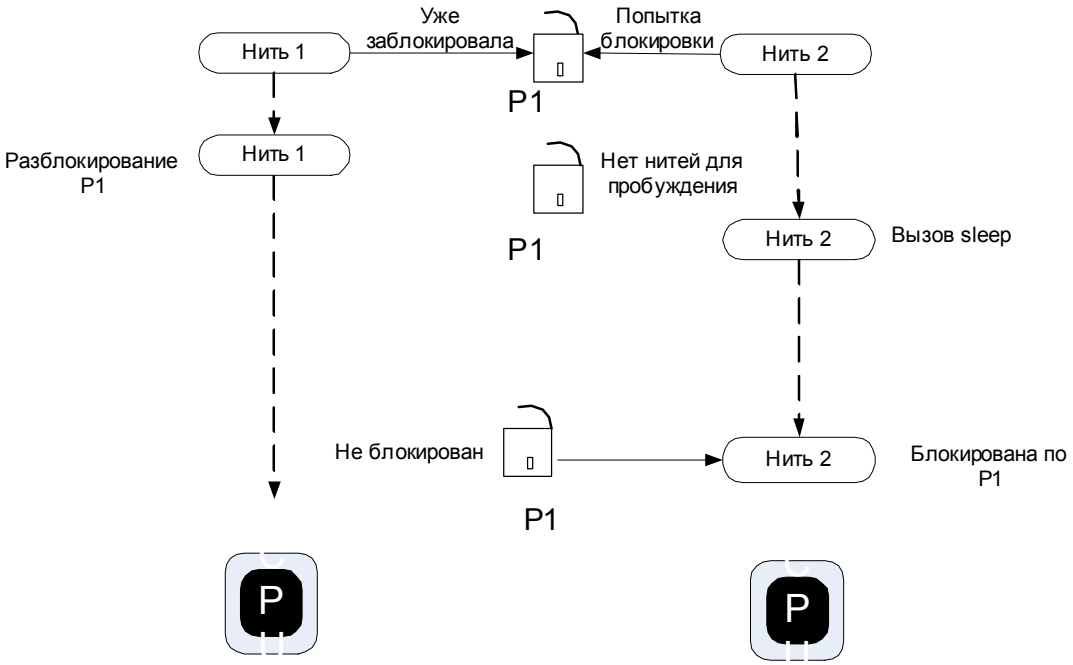
Еще одним примером необходимости изменений является приостановка обработки прерываний. В многопроцессорных системах нить обычно имеет право блокировать прерывания, только на том процессоре, на котором она выполняется. Чаще всего не существует возможности приостановки прерываний на всех CPU системы, так как некоторые процессоры могут уже обрабатывать прерывания, приводящие к возникновению конфликта. Возможно повреждение структур данных ядра, когда обработчик прерывания выполняется на другом CPU. Еще одной проблемой является невозможность использования обработчиком модели синхронизации, основанной на приостановке и возобновлении выполнения, так как большинство реализаций ОС не позволяет обработчикам производить блокировку нитей. Система должна поддерживать некий механизм, позволяющий

приостанавливать обработку прерываний, возникающих на других процессорах. Одним из возможных решений проблемы является использование глобальных уровней ***ipl***, поддерживаемых на программном уровне.

Проблема выхода из режима ожидания

В многопроцессорных системах механизм ожидания/возобновления выполнения работает некорректно. Пример возникновения состязательности при применении этой технологии показан на рисунке (см. *Рисунок 5.4*).

Рисунок 5.4 Проблема выхода из режима ожидания



Пусть нить 1 заблокировала ресурс P1. Нить 2, выполняющаяся на другом процессоре, пытается получить тот же ресурс, но обнаруживает, что он уже занят. Тогда нить 2 вызывает функцию **`sleep()`**, чтобы перейти в режим ожидания ресурса P1. Между операциями проверки занятости ресурса и вызовом **`sleep()`** происходит освобождение P1, при этом нить 1 инициирует пробуждение всех нитей, находящихся в режиме ожидания этого ресурса. Так как нить 2 еще не успела переместиться в очередь ожидания, она не может получить сигнал о необходимости возобновления работы. В результате возникает ситуация, когда ресурс уже свободен, но при этом нить 2 продолжает ожидать его разблокирования. Если в дальнейшем ни одна нить не запросит ресурс P1, то нить 2 будет оставаться в режиме ожидания

постоянно. Для ее решения необходимо объединить процедуру проверки ресурса на занятость и последующий вызов **`sleep()`** в единую, неделимую операцию.

Приведенный пример показывает необходимость использования других механизмов работы с ресурсами для многопроцессорных систем.

Проблема быстрого роста

Когда нить освобождает ресурс, происходит пробуждение всех остальных нитей, которые ожидали освобождения этого ресурса. При этом только одна нить может занять ресурс, а остальные нити увидят, что он опять занят и снова перейдут в режим ожидания.

Эта не проблема для однопроцессорных систем, так как в каждый момент активна только одна нить, которая может освободить ресурс. В многопроцессорной системе может возникнуть ситуация, когда несколько нитей, ожидавших освобождения ресурса, назначаются планировщиком на выполнение одновременно на различных процессорах. Это приводит к попытке захвата ресурса всеми этими нитями. Такая ситуация называется ***проблемой быстрого роста***.

Даже в случае если только одна нить оказалась заблокированной в ожидании ресурса, все равно между изменением состояния нити (готова к выполнению/выполняется) проходит какое то время. За этот промежуток времени ресурс может занять другая нить и это приведет к блокировке первой нити.

Семафоры

Синхронизация в первых реализациях системы UNIX для многопроцессорных систем строилась на использовании семафоров Дейкстры. Их также называют семафорами со счетчиком (counted semaphores).

Семафоры - это объекты, находящиеся в диапазоне целых чисел, которые поддерживают две операции, P() и V(). Примитив P() применяется для декремента (уменьшения на единицу) значения семафора, если новое значение оказывается меньше нуля, то он блокируется. Операция V() используется для инкремента значения, при этом если результат оказывается равным нулю или больше, то V() пробуждает нить или процесс. Эти операции являются неделимыми.

Семафоры можно применять для реализации протоколов синхронизации. Например, представьте, какие проблемы возникают при управлении исчисляемого ресурса (обладающего определенным числом элементов). Процесс пытается получить одну из составляющих ресурса и освободить ее после завершения использования. Ресурс можно представить семафором, который применятся для управления доступа к нему. Примитив P() применятся при каждой попытке запроса ресурса, и уменьшает значение семафора при ее удачном завершении. Если переменная станет, равна нулю (т. е. свободных ресурсов больше нет), все последующие операции приведут к блокировке. При освобождении ресурса стартует операция V(), которая увеличивает значение семафора, что приводит к пробуждению заблокированного процесса.

Во многих системах UNIX семафоры применяются ядром для синхронизации внутренних операций. Они также используются в этом качестве и для пользовательских приложений.

Семафоры System V

В ОС System V поддерживается наиболее общая версия семафоров. Системный вызов **semget** создает или запрашивает массив семафоров (верхнюю границу которого можно назначить). Синтаксис вызова таков:

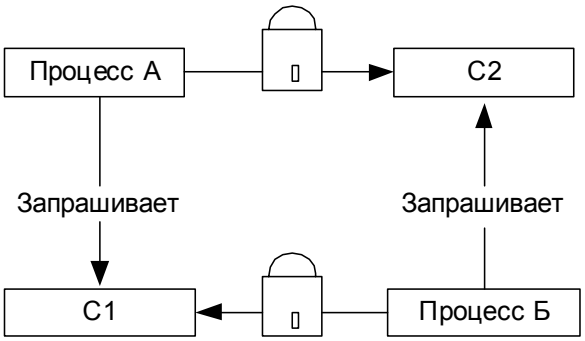
```
semid = semget(key, count, flag);
```

где **key** - 32-разрядная переменная, передаваемая вызывающим процессом. Функция **semget** возвращает массив **count** семафоров, ассоциированных с ключом (**key**). Если с ключом не связано ни одного набора семафоров, то вызов будет возвращать ошибку до тех пор, пока не будет задан флаг **IPC_CREAT**, создающий новый набор семафоров. Если функции передается флаг **IPC_EXCL**, то **semget** возвращает ошибку в том случае, если набор семафоров для указанного ключа уже существует. Переменная semid применяется в последующих операциях над семафорами, она идентифицирует массив семафоров.

Основными проблемами, связанными с применением семафоров, являются условия состязательности и предупреждение взаимоисключений. Использование одиночных семафоров (вместо их массивов) может привести к взаимной блокировке в том случае, если процессу необходимо запросить несколько семафоров. Например, на рисунке (*Рисунок 5.5*) процесс А, удерживающий семафор С1, пытается получить семафор С2 в то время как, процесс Б уже владеет семафором С2 и стремится к тому же в отношении С1. В таком случае ни один из процессов не сможет продолжить функционирование. Хотя это

простейший случай легко обнаружить (и, следовательно, защититься от его возникновения), клинч может произойти и в более сложных вариациях, затрагивающих не один процесс и семафор.

Рисунок 5.5 Возникновение взаимной блокировки



Коды обнаружения взаимоблокировок и защиты от них реализовывать внутри ядра системы непрактично. Более того, не существует общих и универсальных алгоритмов, защищающих от всевозможных ситуаций. Из этого можно сделать вывод, что ядро оставляет решение проблемы взаимоблокировок приложениям. Предлагая механизм наборов семафоров и неделимые операции над ними, ядро системы дает разработчику интеллектуальные механизмы обработки семафоров. Программисты могут выбрать несколько известных методов защиты от тупиковых ситуаций.

Простая блокировка

Простейшим элементом блокировки является объект, который называют ***циклической блокировкой*** (***spin lock***) или ***взаимным исключением*** (***simple mutex***). Если ресурс защищен с помощью этого механизма, то нить, которая пытается получить к нему доступ, перейдет в режим занятого ожидания (выполнение цикла) до тех пор, пока объект не освободится. Взаимоблокировка представляет собой переменную, принимающую значение 0, если ресурс свободен, и значение 1, если он занят. Значение переменной определяется внутри цикла при помощи неделимой команды тестирования и установки.

Самое важное свойство простой блокировки, это продолжение выполнения нити (использование ею процессорного времени) во время ожидания освобождения ресурса. Такую блокировку можно применять только для коротких промежутком ожидания.

Что лучше?

Пусть ресурс защищен при помощи семафора. В этом случае, если нить пытается получить доступ к объекту, а он оказывается занятым, ее выполнение приостанавливается в ожидании освобождения ресурса.

Ругой вариант, это ожидание освобождения ресурса в цикле.

Как выбрать какой вариант лучше?

В режиме ждущего цикла занимается процессор. А в режиме приостановки и возобновления нити требуется переключение контекста и работа с очередями сна и планировщика. Неразумно:

- ждать ресурс, который будет занят в течение продолжительного времени;
- и приостанавливать выполнение нити в ожидании объекта, который скоро освободится.

Следовательно, ни ждущий цикл, ни блокировка не являются идеальным выходом. Одним из решений проблемы является использование двух вариантов синхронизации. Нужно указать нити как нужно поступить:

- приостановить работу;
- или выполнить цикл ожидания.

ЛАБОРАТОРНАЯ РАБОТА 5

СИНХРОНИЗАЦИЯ

Теория

Для синхронизации доступа нескольких процессов к разделяемым ресурсам, используются ***семафоры***. Семафоры не предназначены для обмена данными, как в FIFO или очереди сообщений. Вместо этого, они позволяют разрешить или запретить процессу, использовать разделяемый ресурс.

Рассмотрим применение семафоров на примере. Допустим, имеется разделяемый ресурс (например, файл). Необходимо блокировать доступ к ресурсу для других процессов, когда некий процесс производит операцию над ресурсом (например, записывает в файл). Для этого свяжем с данным ресурсом некую целочисленную величину - счетчик, доступный для всех процессов. Пусть, значение счетчика 1 означает доступность ресурса, а 0 - его недоступность. Тогда перед началом работы с ресурсом процесс должен проверить значение счетчика. Если оно равно 0 - ресурс занят и операция недопустима - процессу остается ждать. Если значение счетчика равно 1 - можно работать с ресурсом. Прежде всего, процессу необходимо заблокировать ресурс, т.е. изменить значение счетчика на 0. После выполнения операции для освобождения ресурса значение счетчика необходимо изменить на 1. В приведенном примере счетчик играет роль семафора.

Для нормальной работы семафора должны быть выполнены следующие условия:

- Значение семафора должно быть доступно различным процессам. Поэтому семафор должен находится не в адресном пространстве процесса, а в адресном пространстве ядра.
- Операция проверки и изменения значения семафора должна быть реализована в виде одной атомарной по отношению к другим процессам (т.е. непрерываемой другими процессами) операции. В противном случае возможна ситуация, когда после проверки значения семафора выполнение процесса будет прервано другим процессом, который в свою очередь проверит семафор и изменит его значение. Единственным способом гарантировать атомарность критических участков операций является выполнение этих операций в режиме ядра.

Семафоры являются системными ресурсами, действия над которым производятся через интерфейс системных вызовов.

Семафоры в System V

Семафоры в System V обладают следующими характеристиками:

- Семафор представляет собой не один счетчик, а группу (набор), состоящую из нескольких счетчиков, объединенных общими признаками (дескриптором объекта, правами доступа и т.д.)
- Каждое из этих чисел может принимать любое неотрицательное значение в пределах, определяемых системой (а не только значения 0 и 1).

Для каждой группы семафоров (в дальнейшем мы будем называть группу просто семафором) ядро поддерживает структуру, определенную в файле **<sys/sem.h>**:

```
struct semid_ds {
  struct ipc_perm sem_perm; // описание прав доступа
  struct sem *sem_base; // указатель на массив семафоров в наборе
  ushort sem_nsems; // количество семафоров в наборе
  time_t sem_otime; // время последнего вызова semop()
  time_t sem_ctime; // время создания
  };
```

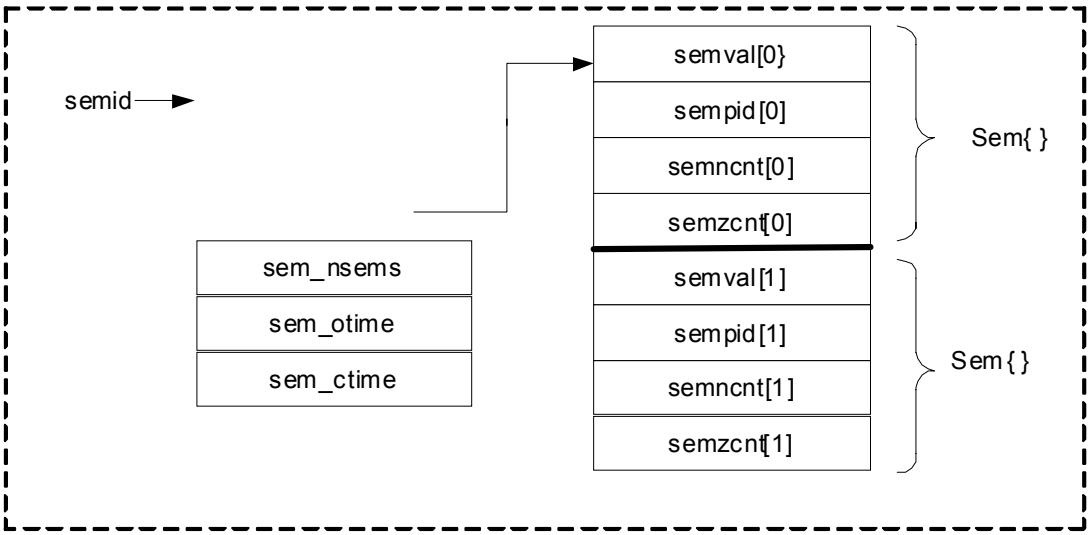
Структура **sem** представляет собой внутреннюю структуру данных, используемую ядром для хранения набора значений семафора. Каждый элемент набора семафоров описывается так:

```
struct sem {
  ushort semval; // значение семафора
  short sempid; // PID процесса, выполнившего последнюю операцию semop
  ushort semncnt;
// количество процессов, ожидающих, что значение семафора превысит текущее
  ushort semzcnt;
// количество процессов, ожидающих, что значение семафора станет равным 0
  };
```

Обратите внимание, что sem_base это указатель на массив структур типа sem (по одному элементу массива на каждый семафор в наборе). Помимо собственно значения семафора, в структуре **sem** хранится идентификатор процесса, вызвавшего последнюю операцию над семафором, число процессов, ожидающих увеличения значения семафора, и число процессов, ожидающих, когда значение семафора станет равным нулю.

Таким образом, любой семафор это - структура **semid_ds**, указывающая на массив структур **sem**.. На рисунке (см. *Рисунок 5.6*) представлен набор из двух элементов (**sem_nsems=2**).

Рисунок 5.6 Массив структур **sem**



Для получения доступа к семафору (и для его создания, если он не существует) используется системный вызов **semget**:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflag);
```

В случае успешного завершения операции функция возвращает дескриптор объекта, в случае неудачи - -1. Аргумент **nsems** задает число семафоров в группе. В случае, когда мы не создаем, а лишь получаем доступ к существующему семафору, этот аргумент игнорируется. Аргумент semflag определяет права доступа к семафору и флажки для его создания (**IPC_CREAT**, **IPC_EXCL**).

После получения дескриптора объекта процесс может производить операции над семафором, подобно тому, как после получения файлового дескриптора процесс может читать или записывать данные в файл. Для этого используется системный вызов **semop**:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf *semop, size_t nops);
```

В качестве второго аргумента функции передается указатель на структуру данных, определяющую операции, которые требуется произвести над семафором с дескриптором **semid**. Операций может быть несколько, и их число указывается в последнем аргументе **nops**.

Важно, что ядро обеспечивает атомарность выполнения критических участков операций (например, проверка значения - изменение значения) по отношению к другим процессам.

Каждый элемент набора операций **semop** имеет вид:

```
struct sembuf{
  short sem_num;/*номер семафора в группе*/
  short sem_op;/*операция*/
  short sem_flg;/*флаги операции*/
  };
```

UNIX допускает три возможные операции над семафором, определяемые **semop**:

- Если величина semop положительна, то текущее значение семафора увеличивается на эту величину. Эта операция изменяет значение семафора (безусловное выполнение).
- Если значение **semop** равно нулю, процесс ожидает, пока семафор не обнулится. Эта операция только проверяет его значение (условное выполнение).
- Если величина **semop** отрицательна, процесс ожидает, пока значение семафора не станет большим или равным абсолютной величине semop. Затем абсолютная величина **semop** вычитается из значения семафора. Эта операция проверяет, а затем изменяет значение семафора (условное выполнение).

При работе с семафорами взаимодействующие процессы должны договориться об их использовании и кооперативно проводить операции над семафорами. ОС не накладывает ограничений на использование семафоров. В частности, процессы вольны решать, какое значение семафора является разрешающим, на какую величину изменяется значение семафора и т.п.

Таким образом, при работе с семафорами процессы используют различные комбинации из трех операций, определенных системой, по-своему трактуя значения семафоров.

В качестве примера рассмотрим два случая использования бинарного семафора (т.е. значения которого могут принимать только 0 и 1).

Пусть в первом примере значение 0 является разрешающим, а 1 запирает некоторый разделяемый ресурс (файл, разделяемая память и т.п.), ассоциированный с семафором. Определим операции, запирающие ресурс и освобождающие его:

```
static struct sembuf sop_lock[2] = {
   0, 0, 0,/*ожидаем обнуления семафора*/
   0, 1, 0,/*затем увеличить значение семафора на 1*/
   };

static struct sembuf sop_unlock[1] = {
   0, -1, 0,/*обнулить значение семафора*/
   };
```

Итак, для запирания ресурса процесс производит вызов, обеспечивающий атомарное выполнение двух операций:

- **Ожидание доступности ресурса.** В случае, если ресурс уже занят (значение семафора равно 1), выполнение процесса будет приостановлено до освобождения ресурса (значение семафора равно 0).
- **Запирание ресурса.** Значение семафора устанавливается равным 1.

```
semop(semid, &sop_lock[0], 2);
```

Для освобождения ресурса процесс должен произвести вызов, который уменьшит текущее значение семафора (равное 1) на 1, и оно станет равным 0, что соответствует освобождению ресурса. Если какой-либо из процессов ожидает ресурса (т.е. произвел вызов операции **sop_lock**), он будет "разбужен" системой, и сможет в свою очередь запереть ресурс и работать с ним.

```
semop(semid, &sop_unlock[0], 1);
```

Во втором примере изменим трактовку значений семафора: значению 1 семафора соответствует доступность некоторого ассоциированного с семафором ресурса, а нулевому значению - его недоступность. В этом случае содержание операций несколько изменится.

```
static struct sembuf sop_lock[2] = {
0, -1, 0,/*ожидать разрешающего сигнала (1), затем обнулить семафор*/
   };

static struct sembuf sop_unlock[1] = {
   0, 1, 0,/* увеличить значение семафора на 1*/
   };
```

В этом случае процесс запирает ресурс вызовом:

```
semop(semid, &sop_lock[0], 1);
```

А освобождает:

```
semop(semid, &sop_unlock[0], 1);
```

Во втором случае операции получились проще (по крайней мере, их код стал компактнее), однако этот подход имеет потенциальную опасность: при создании семафора, его значения устанавливаются равными 0, и во втором случае он сразу же запирает ресурс. Для преодоления данной ситуации процесс, первым создавший семафор, должен вызвать операцию **sop_unlock**, однако в этом случае процесс инициализации семафора перестанет быть атомарным и может быть прерван другим процессом, который, в свою очередь, изменит значение семафора. В итоге, значение семафора станет равным 2, что повредит нормальной работе с ресурсом.

Можно предложить следующее решение данной проблемы:

```
/*Создаем семафор*/
/*если он уже существует semget возвращает ошибку*/
/*поскольку указан флаг IPC_EXCL*/

if((semid = semget(key, nsems, perms|IPC_CREAT|IPC_EXCL)
{
  if(errno == EEXIST)
  {
  /*Действительно, ошибка вызвана существованием объекта*/

  if((semid = semget(key, nsems, perms))<0)
    return(-1); /*Возможно, не хватает системных ресурсов*/
  }
else return(-1); /*Возможно, не хватает системных ресурсов*/
}

/*Если семафор создан нами, проанализируем его*/

else semop(semif, &sop_unlock[0], 1);
```

Чему нужно научиться?

Изучить использование для синхронизации семафоров System V и семафоров Posix.

Задание

Уровень 1 (A)

Посмотрите на сообщение “**Ошибка! Источник ссылки не найден**” и **разберитесь, что там происходит**.

Пояснения: Одновременно могут работать несколько клиентов. Сервер ожидает начала работы какого-либо клиента, после чего ждет

освобождения разделяемой памяти, блокирует ее и читает сообщение. Затем сервер пишет сообщение в разделяемую память и освобождает ресурс. Сервер корректно завершает свою работу при получении сигнала **SIGINT** (он удаляет созданные ранее семафоры и область разделяемой памяти, затем завершается сам). Клиентов несколько и можно организовать их "одновременное" выполнение.

Последовательность действий:

- откомпилировать программы (**gcc -o server server.c; gcc -o client client.c**);
- запустить на выполнение исполняемый модуль **./server**;
- запустить на выполнение исполняемый модуль **./client**;
- проверить, все ли работает правильно;
- послать процессу **server** сигнал **SIGINT**.

Уровень 2, 3 (А)

Необходимо реализовать задачу производители/потребители. Используется циклический буфер (заполнив последнее поле, производитель переходит к началу буфера и заполняет первое поле). Потребитель не должен опережать производителя. Используются три семафора:

- Бинарный семафор **mutex** защищает критические области кода (помещение данных в буфер для производителя и изъятие данных из буфера для потребителя);
- Семафор-счетчик **nempty** подсчитывает количество свободных полей в буфере. Он инициализируется значением **NBUFF** (размер буфера);
- Семафор-счетчик **nstored** подсчитывает количество заполненных полей в буфере. Он инициализируется значением 0).

Скелет кода для задания 1 (А)

shmem.h

```
#define MAXBUFF 80 /максимальная длина сообщения в разделяемой памяти
#define PERM 0666 /права доступа к разделяемой памяти

typedef struct mem//структура данных в разделяемой памяти
{
   int segment;
   char buff[MAXBUFF];
}Message;
```

```
//ожидание начала выполнения клиента
static struct sembuf proc_wait[1] = {1, -1, 0};
//уведомление сервера о том, что клиент начал работу
static struct sembuf proc_start[1] = {1, 1, 0};
//блокирование разделяемой памяти
static struct sembuf mem_lock[2] = {0, 0, 0, 0, 1, 0};
//освобождение ресурса
static struct sembuf mem_unlock[1] = {0, -1, 0};
```

server.c

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include "shmem.h"

Message* msgptr;
int shmid, semid;

void hndlr(int sig)//обработчик сигнала SIGINT
{
  signal(SIGINT, hndlr);
  //отключение от области разделяемой памяти
  if(shmdt(msgptr) < 0)
  {
    printf("Server: error\n");
    exit(-1);
  }
  //удаление созданных объектов
  if(shmctl(shmid, IPC_RMID, 0) < 0)
  {
    printf("Server : can't delete area\n");
    exit(-1);
  }
  printf("Server: area is deleted\n");
  if(semctl(semid, 0, IPC_RMID) < 0)
  {
    printf("Server : can't delete semaphore\n");
    exit(-1);
  }
  printf("Server: semaphores are deleted\n");

}

void main(void)
{
  key_t key;
  signal(SIGINT, hndlr);
  //получение ключа как для семафора так и для разделяемой памяти
  if((key = ftok("/home/matveev/test.txt", 'A')) < 0)
  {
    printf("Server: can't get a key\n");
```

```
    exit(-1);
  }
  //создание области разделяемой памяти
  if((shmid = shmget(key, sizeof(Message), PERM | IPC_CREAT)) < 0)
  {
    printf("Server: can't create an area\n");
    exit(-1);
  }
  printf("Server: area is created\n");
  //присоединение области
  if((msgptr = (Message*)shmat(shmid, 0, 0)) < 0)
  {
    printf("Server: error of joining\n");
    exit(-1);
  }
  printf("Server: area is joined\n");
  //создание группы из 2 семафоров
  //1 - для синхронизации работы с разделяемой памятью
  //2 - для синхронизации выполнения процессов
  if((semid = semget(key, 2, PERM | IPC_CREAT)) < 0)
  {
    printf("Server: can't create a semaphore\n");
    exit(-1);
  }
  printf("Server: semaphores are created\n");
  while(1)
  {
    //ожидание начала работы клиента
    if(semop(semid, &proc_wait[0], 1) < 0)
    {
      printf("Server: execution complete\n");
      exit(-1);
    }
    //ожидание завершения работы клиента с разделяемой памятью
    if(semop(semid, &mem_lock[0], 2) < 0)
    {
      printf("Server: can't execute a operation\n");
      exit(-1);
    }
    //вывод сообщения, записанного клиентом в разделяемую память
    printf("Server: read message\n%s", msgptr->buff);
    //запись сообщения в разделяемую память
sprintf(msgptr->buff,"Message from server with PID = %d\n", getpid());
    //освобождение ресурса
    if(semop(semid, &mem_unlock[0], 1) < 0)
    {
      printf("Server: can't execute a operation\n");
      exit(-1);
    }
  }
}
```

client.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include "shmem.h"

void main(void)
{
  Message* msgptr;
  key_t key;
  int shmid, semid;
  //получение ключа
  if((key = ftok("/home/matveev/test.txt", 'A')) < 0)
  {
    printf("Client: can't get a key\n");
    exit(-1);
  }
  //получение доступа к разделяемой памяти
  if((shmid = shmget(key, sizeof(Message), 0)) < 0)
  {
    printf("Client: access denied\n");
    exit(-1);
  }
  //присоединение разделяемой памяти
  if((msgptr = (Message*)shmat(shmid, 0, 0)) < 0)
  {
    printf("Client : error of joining\n");
    exit(-1);
  }
  //получение доступа к семафору
  if((semid = semget(key, 2, PERM)) < 0)
  {
    printf("Client: access denied\n");
    exit(-1);
  }
  //блокировка разделяемой памяти
  if(semop(semid, &mem_lock[0], 2) < 0)
  {
    printf("Client : can't execute a operation\n");
    exit(-1);
  }
  //уведомление сервера о начале работы
  if(semop(semid, &proc_start[0], 1) < 0)
  {
    printf("Client : can't execute a operation\n");
    exit(-1);
  }
  //запись сообщения в разделяемую память
  sprintf(msgptr->buff,"Message from client with PID = %d\n", getpid());
  //освобождение ресурса
  if(semop(semid, &mem_unlock[0], 1) < 0)
  {
    printf("Client: can't execute a operation\n");
```

```
    exit(-1);
  }
  //ожидание завершения работы сервера с разделяемой памятью
  if(semop(semid, &mem_lock[0], 2) < 0)
  {
    printf("Client: can't execute a operation\n");
    exit(-1);
  }
  //чтение сообщения из разделяемой памяти
  printf("Client: read message\n%s", msgptr->buff);
  //освобождение разделяемой памяти
  if(semop(semid, &mem_unlock[0], 1) < 0)
  {
    printf("Client: can't execute a operation\n");
    exit(-1);
  }
  //отключение от области разделяемой памяти
  if(shmdt(msgptr) < 0)
  {
    printf("Client: error\n");
    exit(-1);
  }
}
```

Скелет кода для задания 2, 3 (А)

```
struct {
  int buff[NBUFF];
  sem_t *mutex, *nempty, *nstored;
  } shared;

void *produce(void *), *consume(void *);

int main(int argc, char **argv)
{
  pthread_t tid_produce, tid_consume;

// создание трех семафоров
  shared.mutex= …;
  shared.nempty=   ;
  shared.nstored=   ;

Pthread_create(tid_produce, NULL);
Pthread_create(tid_consume, NULL);

Pthread_join(tid_produce, NULL);
Pthread_join(tid_consume, NULL);

Sem_unlink(…);
Sem_unlink(…);
Sem_unlink(…);
exit(0);
}
```

```
void *produce(void *arg)
{
  int i;

  for(i=0; i<nitems; i++)
  {
    Sem_wait(shared.nempty);
    Sem_wait(shared.mutex);
    // помещаем i в циклический буфер
    Sem_post(shared.mutex);
    Sem_post(shared.nstored);
  }
  return(NULL);
}

void *consume(void *arg)
{
  int i;

  for(i=0; i<nitems; i++)
  {
    Sem_wait(shared.nstored);
    Sem_wait(shared.mutex);
    // ожидаем появления объекта в буфере
    Sem_post(shared.mutex);
    Sem_post(shared.nempty);
  }
  return(NULL);
}
```

Глава

6

Виртуальная память

Введение

Одной из главных функций ОС является эффективное ***управление памятью***. В каждом компьютере есть высокоскоростная память с произвольным доступом (оперативная память, RAM), ее называют основной памятью, физической памятью, первичной памятью или просто памятью. Скорость доступа к оперативной памяти составляет несколько тактов процессора. Программы могу напрямую обращаться к данным или участкам кода, находящимся только в оперативной памяти компьютера. Микросхемы памяти RAM достаточно дорогие, а памяти всегда не хватает. Для хранения данных, которые не помещаются в основную память, обычно используются жесткие диски (вторичная память). Скорость обращения к таким устройствам ниже по сравнению со скоростью доступа к оперативной памяти, а операции доступа требуют определенных усилий ОС. В ОС за распределение информации между оперативной памятью и вторичными устройствами хранения отвечает ***подсистема управления памятью*** ядра. Ядро взаимодействует с ***блоком управления памятью*** (***memory management unit***, ***MMU***), который отвечает за получение данных из оперативной памяти и помещение данных в нее.

Если ОС не надо управлять памятью, то ее жизнь становится значительно проще. В этом случае системе надо поддерживать выполнение только одной программы, которая постоянно загружена по фиксированному адресу в памяти. Эта программа получает в свое распоряжение все ресурсы, а используемые адреса будут представлять собой просто физические адреса, Это наиболее быстрый и эффективный способ выполнения единственной программы.

Этот способ применяется в системах реального времени, основанных на небольших микропроцессорах (например, программа "прошита" в ПЗУ). Рассмотрим недостатки такого подхода:

- Во-первых, размер программы ограничен объемом памяти, поэтому нельзя выполнять большие программы;
- Во-вторых, когда в память загружена одна программа, в случае ожидания ввода/вывода система простаивает.

Если система разработана для поддержки выполнения одной небольшой программы, то, она будет непригодна для многозадачных сред.

Многозадачная ОС должна заниматься управлением памятью и обеспечивать:

- Выполнение программ, размер которых больше, чем объем физической памяти. В идеале система должна обеспечить выполнять программы любого размера;
- Выполнение частично загруженных программ. Это уменьшает время их первоначальной загрузки;
- Размещение в памяти одновременно более чем одной программы. Это увеличивает коэффициент использования процессора;
- Возможность выполнения перемещаемых в памяти программ, которые могут располагаться в любом месте памяти и перемещаться из одного места в другое в течение выполнения;
- Поддержка совместного использования ресурсов, например, если два процесса загружают одну и ту же программу, они должны иметь возможность использовать сообща ее код.

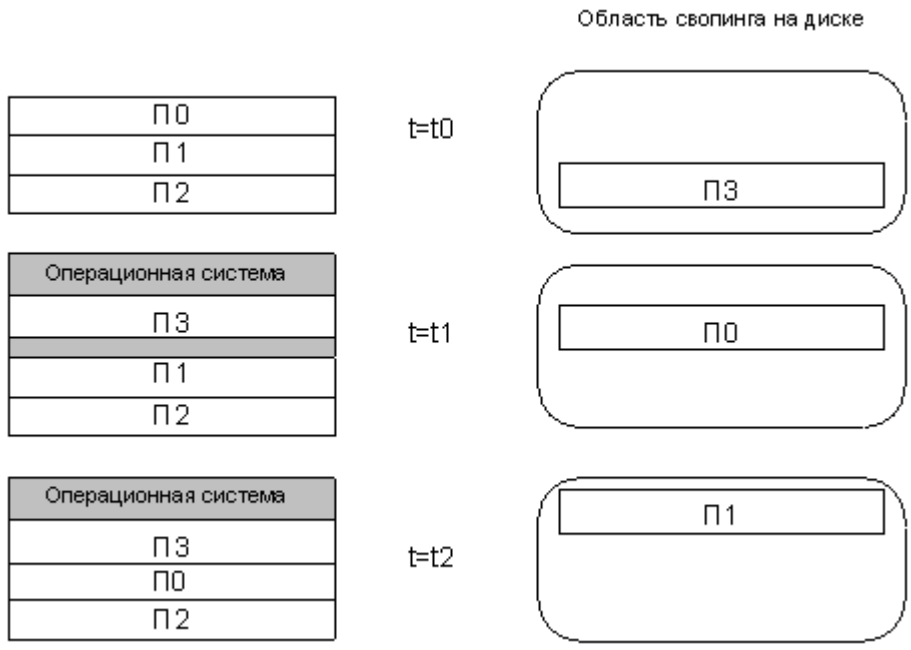
Система ***виртуальной памяти*** как раз и решает поставленные задачи. Приложению становится доступен большой объем памяти, хотя реально компьютер может иметь совсем мало оперативной памяти. Вводится абстракция ***виртуальное адресное пространство***. Программа использует ссылки на коды и данные в своем виртуальном адресном пространстве, а эти адреса должны преобразовываться в адреса ячеек оперативной памяти. Передача информации в основную память для использования ее программой и выполнение трансляции адресов при каждом доступе к памяти требуют совместных действий ОС и аппаратуры.

За поддержку виртуальной памяти приходится расплачиваться:

- Во-первых, таблицы трансляции адресов и другие структуры данных, создаваемые для управления памятью, надо тоже хранить в памяти, а из-за этого уменьшаются количество физической памяти, доступное программам;
- Во-вторых ко времени выполнения каждой инструкции прибавляются затраты на преобразование адресов. Если процесс пытается получить доступ к странице, не загруженной в основную память, то система поместит эту страницу в оперативную память, а это потребует относительно медленных операций дискового ввода-вывода. Все действия с памятью занимают значительную часть процессорного времени (примерно 10% на загруженных системах).

Механизмы управления памятью в ранних версиях UNIX ограничивались возможностями ***свопинга*** (***swapping***). Процессы загружались в оперативную память целиком друг за другом. В один момент времени в физической памяти машины размещалось некоторое количество процессов, а система являлась для них совместно используемым ресурсом (см. *Рисунок 6.1*).

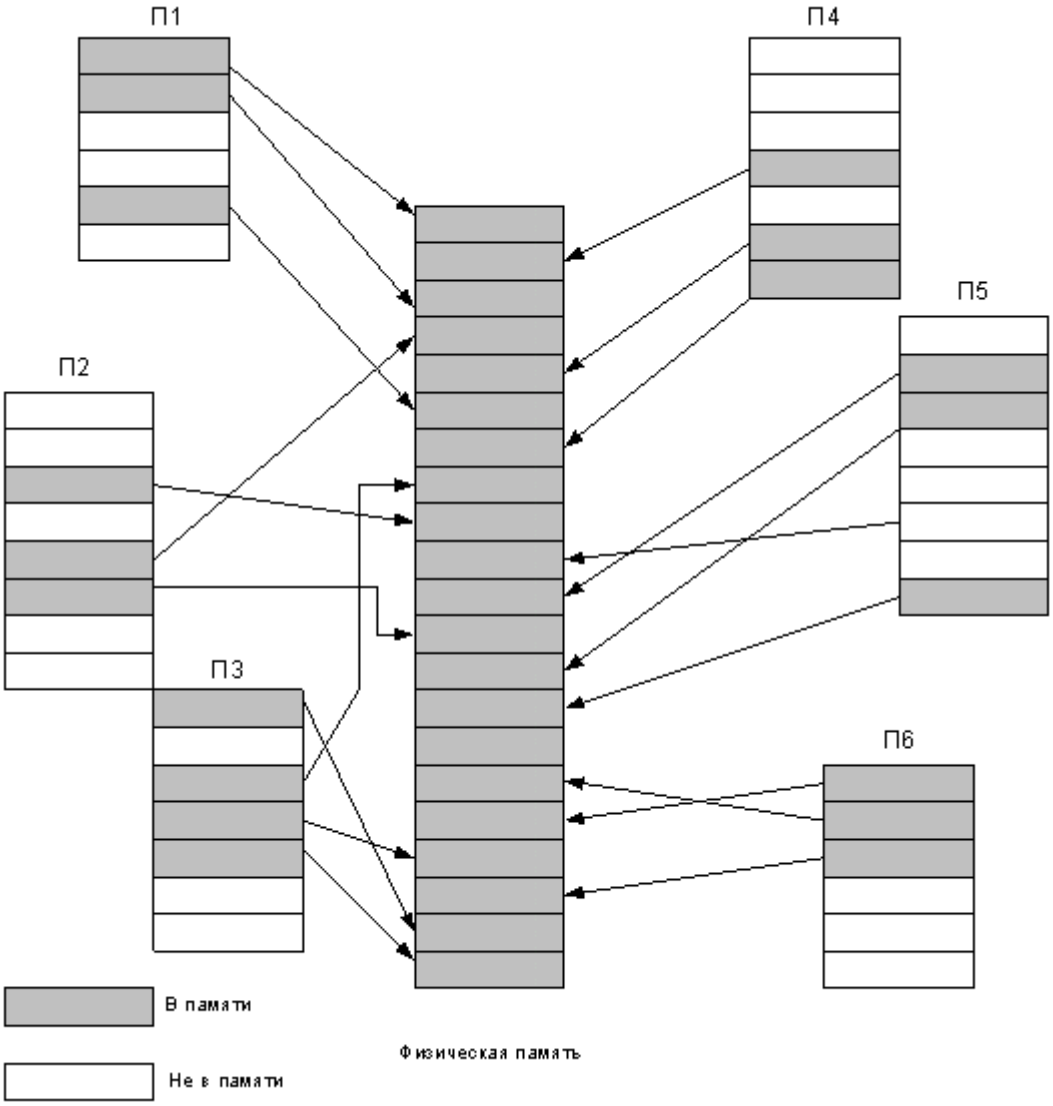
Рисунок 6.1 Свопинг



Если требовалось выполнить другой процесс, то один из существующих процессов выгружался из памяти. Выгружаемый процесс копировался в заранее определенную ***область свопинга*** (***swapping partition***), расположенную на диске. Некоторое ***пространство свопинга*** (***swap space***) выделялось для каждого процесса при его создании, поэтому область свопинга при необходимости была доступна.

Технология *загрузки страниц по запросу* (***demand paging***) появилась в системе UNIX после создания VAX-11/780 в 1978 году. В системах загрузки страниц по запросу память и адресное пространство процесса поделены на страницы фиксированного размера, которые помещаются в память или выгружаются по мере необходимости. Страницу физической памяти часто называют ***страничным фреймом*** (***page frame***) или ***физической страницей*** (***physical page***). В один момент времени могут выполняться сразу несколько процессов, при этом в физической памяти располагается лишь часть страниц каждого из них (см. *Рисунок 6.2*).

Рисунок 6.2 Страницы и страничные фреймы



Каждая программа считает себя единственной в системе. Виртуальный адрес адреса состоит из номера страницы и смещение на этой странице. Аппаратная часть совместно с ОС преобразует номер виртуальной страницы в номер физического страничного фрейма и затем обращается к соответствующей ячейке памяти. Если требуемой страницы нет в оперативной памяти, ее необходимо туда загрузить. В идеальном случае ни одна из страниц не помещается в память до тех пор, пока она не потребуется. Большинство современных систем UNIX заранее загружают те страницы, которые, по мнению системы скоро потребуются.

Загрузка страниц по запросу может использоваться вместе с технологией свопинга, или отдельно. Перечислим некоторые преимущества загрузки страниц по запросу:

- Размер программы ограничивается только максимальным объемом виртуальной памяти, который на 32-разрядных машинах равен 4 гигабайтам;
- Начальная загрузка программы происходит быстро, поскольку для начала работы программы не требуется полностью помещать ее в память;
- Одновременно в системе может быть загружено сразу несколько программ, так как в определенный момент времени лишь некоторая часть страниц каждой из них должна находиться в памяти;
- Перемещение отдельных страниц памяти намного проще для системы, чем свопинг процессов или сегментов целиком.

Требования к системе виртуальной памяти

Рассмотрим набор требований к системе виртуальной памяти:

- **Управление адресным пространством.** Ядро должно выделять адресное пространство процессу на этапе выполнения **`fork`** и освобождать его при вызове **`exit`**. Если процесс производит системный вызов **`exec`**, ядро переписывает его старое адресное пространство кодом новой программы. Другие важные операции в адресном пространстве подразумевают изменение областей данных или стека, а также добавление новых областей (например, разделяемой памяти).
- **Преобразование адресов.** При выполнении любой инструкции, производящей доступ к памяти, необходимо подставлять вместо виртуальных адресов физические адреса оперативной памяти. В

системах загрузки страниц по запросу страница представляет собой единицу памяти, к которой относятся операции размещения, защиты и преобразования адресов. Виртуальный адрес включает номер виртуальной страницы и смещение внутри этой страницы. Номер виртуальной страницы транслируется в физический номер страницы при помощи карт трансляции адресов. Если процесс попытается получить доступ к странице, отсутствующей в физической памяти, в системе возникнет исключительное состояние ***страничная ошибка*** (***page fault***). Обработчик исключительных состояний ядра исправит эту ошибку, поместив необходимую страницы в оперативную память.

- **Управление физической памятью.** Физическая память это наиболее важный ресурс компьютера, который контролирует подсистема управления памятью. Память необходима как самой ОС, так и прикладным процессам. Суммарное адресное пространство всех активных процессов системы, как правило, превышает объем физической памяти, в которой может находиться лишь ограниченный набор данных;

- **Защита памяти.** ОС должна защищать свои коды и данные от прикладных процессов. В противном случае пользовательская программа может случайно (или специально) повредить ОС. Процессы не должны иметь доступ к страницам, относящимся к другим процессам. Часть адресного пространства процесса должна быть защищена даже от него самого. Например, область кода процесса защищена от записи, иначе процесс может повредить сам себя;

- **Разделение памяти.** Необходимо обеспечить взаимодействие процессов. Например, все процессы одной программы могут разделять между собой единственную копию кода программы. Процессы могут совместно использовать области памяти вместе с другими взаимодействующими процессами. Таким образом можно разделять коды стандартных библиотек. Приведенные примеры относятся к совместному использованию ресурсов на высоком уровне. Но существует и вариант разделения на низком уровне, относящийся к отдельным страницам памяти. Например, после выполнения `fork` предок и потомок могут разделять между собой единственную копию страниц данных и стека до тех пор, пока процессы не попытаются внести в них изменения.

- **Отслеживание загрузки системы.** Страничная система должна удовлетворять потребности активных процессов, но это ей удается не всегда. Может получиться так, что процессы не получат достаточного количества памяти для размещения своих активных страниц и в

результате этого не смогут выполняться дальше. Загруженность страничной системы зависит от общего числа и размеров адресных пространств активных процессов, а также от конфигурации памяти. ОС должна следить за страничной подсистемой с целью обнаружения таких ситуаций и уметь их разрешать. Например, в случае перегрузки система может запретить выполнения новых процессов или завершить выполнение некоторых выполняющихся процессов.

Виртуальное адресное пространство

К виртуальному адресному пространству процесса относятся все (виртуальные) ячейки памяти, к которым может обращаться программа. В любой момент времени виртуальное адресное пространство процесса и его аппаратный контекст отражают текущее состояние процесса. Мы уже рассматривали, что при использовании архитектуры выделения страниц по запросу виртуальное адресное пространство процесса делится страницы. Страницы могут содержать следующую информацию:

- код (текст);
- инициализированные данные;
- неинициализированные данные;
- измененные данные;
- стек;
- кучу;
- разделяемую память;
- разделяемые библиотеки.

Текстовые данные обычно доступны только для чтения. Остальные данные обычно доступны для чтения/записи. Защита совместно используемых страниц задается при выделении соответствующей области памяти.

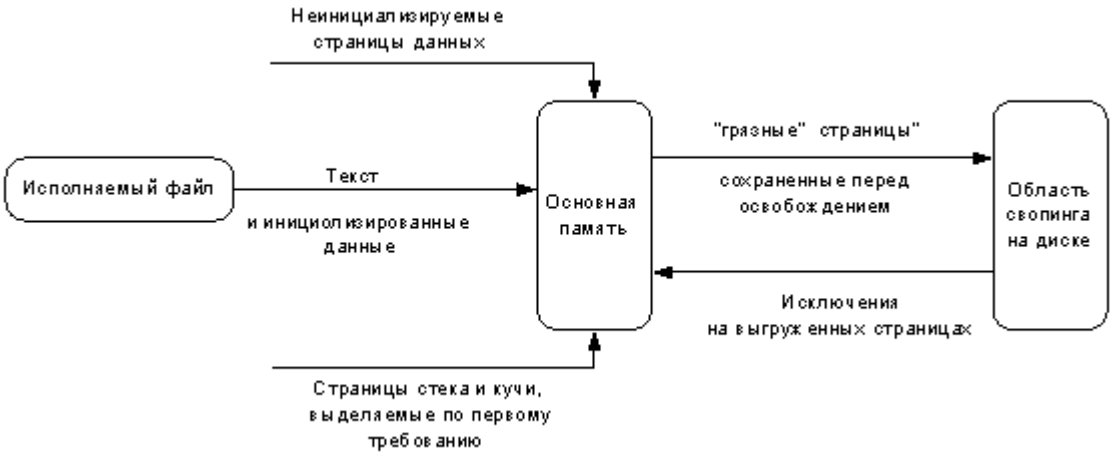
Текстовые страницы разделяются между всеми процессами, выполняющими одну программу. Страницы в разделяемой области памяти используются всеми процессами, которые присоединили эту область к своему адресному пространству. Совместно используемая библиотека может содержать страницы и с текстом и с данными. Текстовые страницы разделяются между всеми процессами, которые используют библиотеку. Страницы данных библиотеки не разделяются,

каждый процесс получает их копию (некоторые реализации позволяют совместно использовать такие страницы до тех пор, пока они не изменяются).

Перемещение страниц

Рассмотрим процесс перемещения страниц (см. *Рисунок 6.3*). При осуществлении доступа к странице, сохраненной в области свопинга, ядро обрабатывает страничную ошибку, читая необходимую страницу из этой области. Для этого ядру необходимо поддерживать карту свопинга, описывающую местонахождение всех страниц, находящихся в области свопинга.

Рисунок 6.3 Схема перемещения страниц

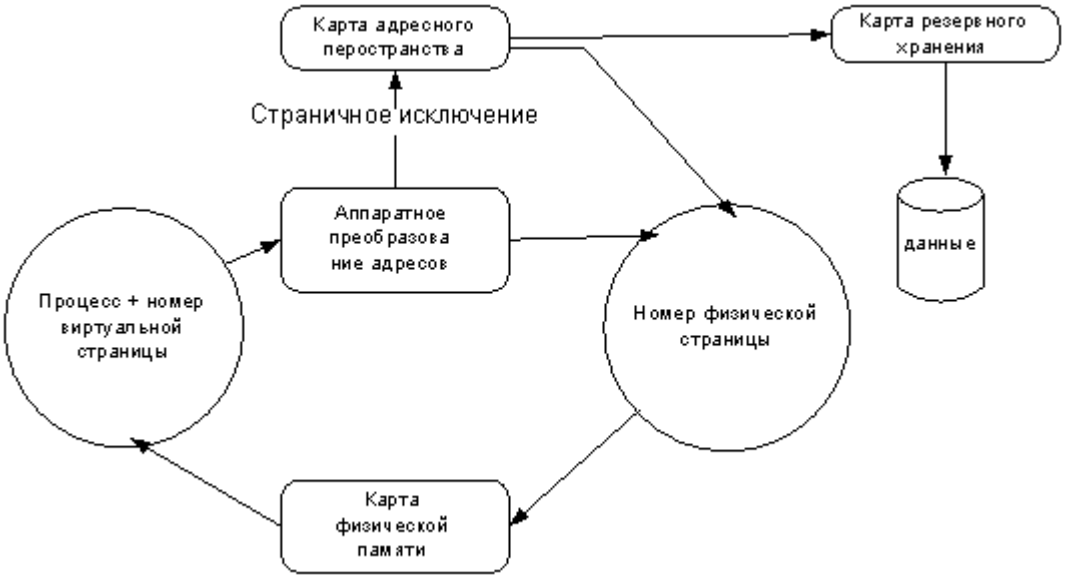


Если страница должна быть удалена из памяти повторно, ее сохранение в области свопинга происходит только в том случае, когда ее содержимое не совпадает с предыдущей копией. Так происходит в случае, когда страница была модифицирована после последнего чтения из области свопинга. Тогда страница является ***"грязной"*** (***dirty***). Следовательно, нужно уметь распознавать подобные страницы. Не нужно хранить в области свопинга текстовые страницы, так как их можно считать прямо из выполняемого файла.

Карты трансляции адресов

Страничная система может использовать четыре различных типа карт трансляции адресов для поддержки виртуальной памяти (см. *Рисунок 6.4*).

Рисунок 6.4 Преобразование адресов



- **Аппаратное преобразование адресов.** Для выполнения каждой команды, осуществляющей доступ к памяти, аппаратная часть должна транслировать виртуальный адрес программы в адрес ячейки физической памяти. Любой компьютер обладает некоторым механизмом аппаратного преобразования адресов.

- **Карта адресного пространства.** Если аппаратная часть не может преобразовать адрес, генерируется ***страничная ошибка*** (***page fault***). Это происходит, если страница не находится в основной памяти или аппаратура не может произвести трансляцию адреса. В этом случае обработчик ошибки ядра, будет помещать нужную страницу в основную память.

- **Карта физической памяти.** Ядру часто нужно производить обратное преобразование и определять процесс, являющийся владельцем данной страницы в оперативной памяти, а также номер виртуальной страницы для нее. Например, при удалении активной страницы из памяти ядру необходимо пометить ее как недействительную. Для этого ядро находит и помечает соответствующее вхождение в ***таблице страниц***.

- **Карта внешней памяти.** Если нужная страница не находится в оперативной памяти, необходимо выделяет страничный фрейм и загрузить страницу из выполняемого файла, объектного файла разделяемой библиотеки или копии, находящейся в области свопинга. Такие страницы составляют внешнюю память по отношению к страницам процесса. Ядро поддерживает карты для нахождения страниц, находящихся во внешней памяти.

Стратегия замещения страниц

Если нет свободных страничных фреймов, то для размещения новой страницы ядро должно удалить страницу, находящуюся в памяти в текущий момент времени. Стратегии замещения страниц служат для принятия решения о том, какую именно страницу следует удалить из памяти. Идеальным кандидатом является ***"мертвая" страница***, которая больше не требуется (например, если она принадлежит завершенному процессу). Если таких страниц нет в памяти (или их мало), ядро может выбрать правило либо ***локального***, либо ***глобального замещения страниц***. Правило локального замещения выделяет определенное количество страниц каждому процессу или группе взаимосвязанных процессов и если процессу необходима новая страница, он должен заменить одну из своих собственных страниц. При ***глобальном замещении*** нужно замещать страницу любого процесса, используя глобальные критерии выбора.

Правила ***локального замещения*** применяются в том случае, если нужно гарантированно выделять ресурсы определенным процессам. Например, системный администратор может выделить больший объем страниц наиболее важным процессам. С другой стороны, ***"глобальные правила"*** проще для реализации и подходят для систем разделения времени. В большинстве вариантов UNIX реализовано правило глобального замещения, но для каждого активного процесса резервируется некое минимальное количество резидентных страниц.

Для реализации глобального замещения необходимо выбрать правило, в соответствии с которым, будет приниматься решение о том, какие страницы хранить в памяти. В идеале нужно хранить только те страницы, которые будут использованы вскоре. Этот набор страниц получил название ***рабочего набора*** (***working set***) процесса.

Обычно для замещения страниц применяется правило ***наименее частого использования*** (***LRU***). При этом удаляются те страницы, к которым доступ давно не производился.

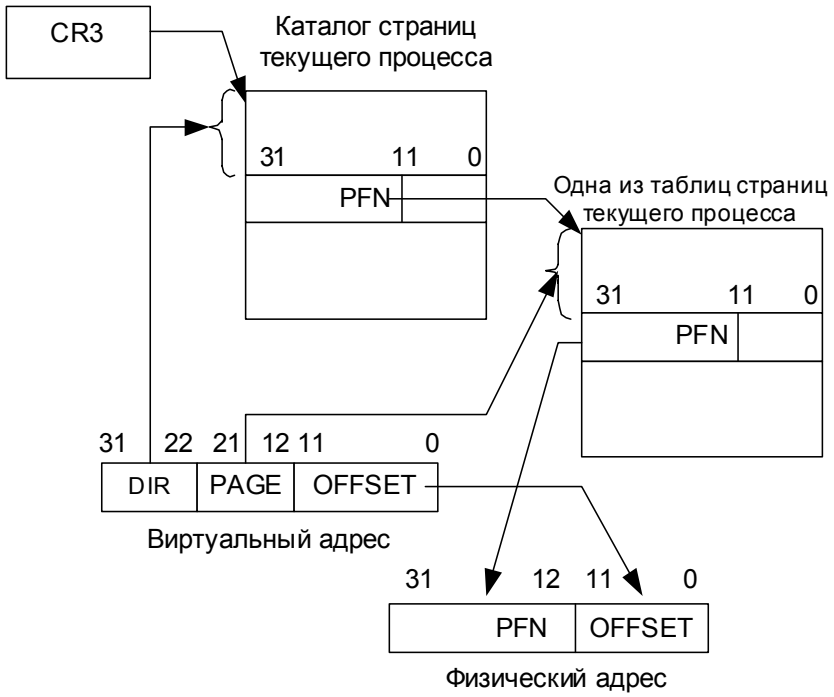
Таблица страниц

В большинстве систем трансляция адресов реализуется при помощи ***таблиц страниц***. Обычно в системе поддерживается одна ***таблица страниц*** для ***адресов ядра*** и одна (или более) таблиц для описания адресного пространства каждого процесса. Таблица страниц представляет собой массив элементов, по одному на виртуальную страницу процесса. ***Элемент таблицы страниц*** (***page table entry***, ***PTE***) определяет описываемую им страницу. Размер одной страницы равен

4 Кбайта. Обычно в однопроцессорной системе активными являются две таблицы, одна для ядра, а вторая для текущего выполняющего процесса.

Архитектура x86 использует ***двухуровневые таблицы страниц*** (см. *Рисунок 6.5*). У каждого процесса имеется ***каталог страниц*** (***page directory***), содержащий элементы, указывающие на таблицы страниц. ***Каталог страниц*** является таблицей страниц первого уровня, а таблицы страниц представляют второй уровень. ***Управляющий регистр CR3*** (его еще называют ***базовым регистром каталога страниц***) содержит номер страничного фрейма текущего каталога страниц.

Рисунок 6.5 Двухуровневая таблица страниц

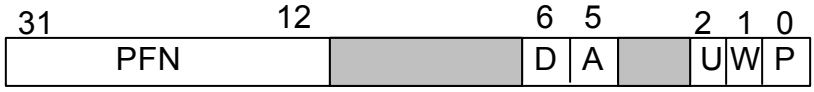


Виртуальный адрес состоит из трех частей:

- старшие 10 разрядов содержат смещение (индекс) от начала ***каталога страниц*** (***DIR***). Складывая содержимое CR3, и это смещение получаем номер строки каталога страниц, в которой содержится адрес таблицы страниц;
- следующие 10 разрядов содержат ***смещение*** (индекс) ***в таблице страниц***. Складывая адрес таблицы страниц и это смещение, получаем номер строки таблицы страниц, в которой содержится номер страничного фрейма;

- младшие 12 разрядов определяют ***смещение на странице***, которое прибавляется к номеру страничного фрейма для получения физического адреса.

Рисунок 6.6 Элемент таблицы страниц



PFN	Номер страничного фрейма
D	“Грязная”
A	Доступна (по ссылке)
U	Пользователь (0)/супервизор (1)
W	Чтение (0)/запись(1)
P	Присутствует (действительна)

Элемент таблицы страниц имеет разрядность 32 бита и состоит из следующих полей (см. *Рисунок 6.6*):

- ***номер страничного фрейма***(31-12);
- ***бит изменения*** (***modified***) - "грязная" (6);
- ***бит ссылки*** (***referenced***) - доступна по ссылке (5)
- ***информация о защите*** (2-1)
  - первый бит:
    - доступность только для чтения - 0;
    - доступность для чтения/записи - 1
  - второй бит:
    - страница пользователя - 0;
    - страница root - 1
- ***бит корректности*** (***valid***) - действительная страница (0).

При каждой записи на страницу аппаратная часть устанавливает ***бит изменения в PTE***. Если бит изменения установлен, то страницу будет сохраняться на диске при уничтожении. Если аппаратура поддерживает ***бит ссылки***, то установка этого бита происходит при каждом обращении к странице.

Где располагаются страницы

В любой момент времени конкретная страница памяти процесса может находиться в одном из следующих состояний:

- ***Резидентная*** (***resident***). Страница загружена в основную память. Элемент таблицы страниц содержит номер странчного фрейма;
- ***Заполненная по необходимости*** (***fill-on-demand***). Процесс пока не произвел ссылку на такую страницу. Она будет помещена в память при первом обращении к ней. Существует два типа страниц, заполняемых при необходимости:
  - ***Заполненная текстом*** (***fill-from-text***). При первом обращении в страницу считываются коды и данные из исполняемого файла;
  - ***Заполненная нулями*** (***zero-fill***). Страница используется для хранения неинициализированных данных, и заполняется нулями при необходимости;
- ***Выгруженная*** (***outswapped***). Страница, которая была считана в память, а затем выгружена в область свопинга для того, чтобы освободить память для размещения другой страницы. Такие страницы могут быть восстановлены из области свопинга.

Отображаемые в память файлы

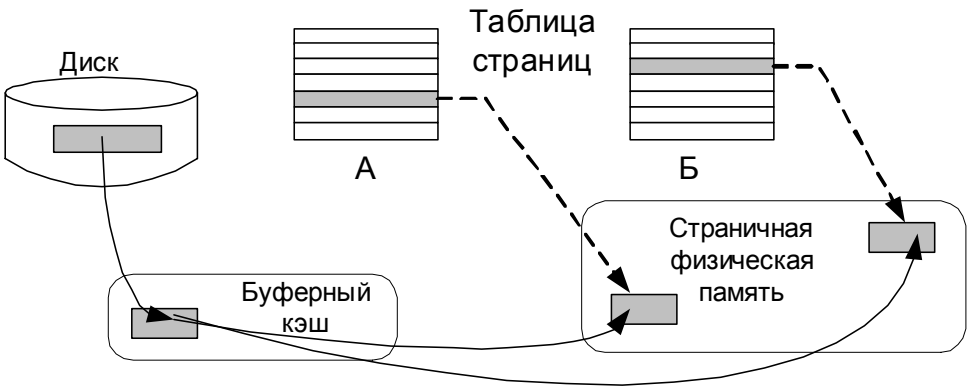
Архитектура управления памятью Virtual Memory (VM) была представлена корпорацией Sun Microsystems в своей системе SunOS 4.0. От системы SunOS требовалась поддержка разделения памяти, совместно используемых библиотек и файлов, отображаемых в память.

Рассмотрим понятие ***отображения файлов*** (***file mapping***). Этот термин используется для описания двух отличных друг от друга сущностей:

- с одной стороны, отображение файлов является средством, позволяющим проецировать файл на часть адресного пространства и после этого использовать простые команды доступа к памяти для осуществления чтения и записи файла;
- с другой стороны, отображение файлов можно использовать в качестве основной схемы организации ядра, которое может видеть все свое адресное пространство в виде набора проекций на объекты различного типа, такие как файлы.

Традиционной методикой работы с файлами в UNIX является их открытие посредством системного вызова open и вызовов функций **`read`**, **`write`**, **`seek`** и **`lseek`** для осуществления последовательного или произвольного ввода-вывода. Это неэффективная методика работы с файлами, так как требуется отдельный системный вызов при каждой необходимости чтения/записи. А если несколько процессов осуществляют доступ к одному и тому же файлу, каждый из них копирует данные этого файла в свое адресное пространство. На рисунке (*Рисунок 6.7*) показана ситуация чтения файла двумя процессами. При этом необходимо произвести чтение с диска для копирования страницы в буферный кэш, а также операции копирования данных из буфера в адресное пространство процесса (количество таких операций равняется количеству процессов, которые будут читать файл). В результате в памяти будет находиться три копии страницы: одна находится в буферном кэше и по одной размещается в адресном пространстве каждого процесса. Для чтения информации каждому процессу нужно отдельно вызвать **`read`** (и дополнительно **`lseek`** для произвольного доступа).

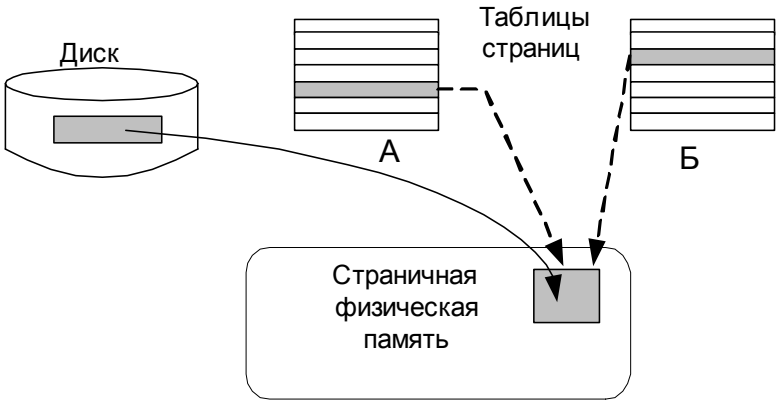
Рисунок 6.7 Чтение файла



Рассмотрим вариант, когда процессы отображают ту же страницу в свои адресные пространства (см. *Рисунок 6.8*).

Ядро создает такое отображение, обновляя структуры управления памятью. Если процесс А пытается обратиться к странице, генерируется страничная ошибка. Ядро исправляет страничную ошибку, считывая страницу в память и модифицируя таблицу страниц так, чтобы в ней появилось ссылка на эту страницу. Если затем процесс Б обратится к этой странице, то она уже находится в памяти и обработка ситуации будет заключаться в изменении ядром таблицы страниц процесса Б.

Рисунок 6.8 Отображение файла



Перечислим преимущества обращения к файлам путем отображения их в память:

- вместо использования двух операции чтения обращаемся к диску один раз;
- после отображения для чтения или записи данных больше не потребуется вызывать системные функции;
- в памяти будет храниться только одна копия страницы.

Что будет, если процесс производит запись в отображенную страницу? Процесс может установить два типа отображения файлов:

- ***Разделяемое*** (***shared***). В этом случае все изменения производятся в самом отображаемом объекте. Ядро передает изменения непосредственно разделяемой копии страницы и сбрасывает их на диск в файле при удалении страницы.
- ***Закрытое*** (***private***). Если отображение является закрытым, то все изменения будут отражены только в закрытой копии страницы, к которой они относятся. При этом операции записи не затрагивают исходный объект, следовательно, ядро не сбрасывает измененные данные в файл при удалении страницы из памяти.

Если несколько процессов разделяют между собой отображение файла, то все изменения на странице, которые делает один из них, сразу же становятся видны остальным процессам.

ЛАБОРАТОРНАЯ РАБОТА 6

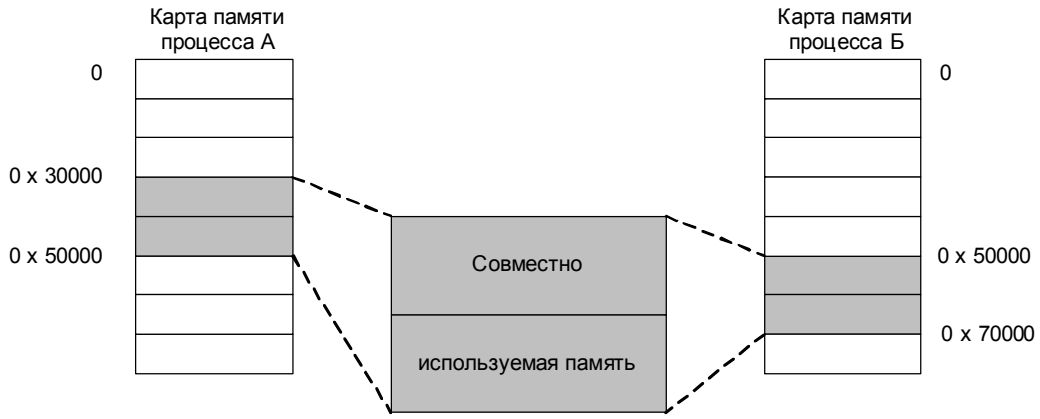
РАЗДЕЛЯЕМАЯ ПАМЯТЬ

Теория

Обмен данными между процессами с использованием механизмов межпроцессорного взаимодействия каналов, FIFO и очередей сообщений может привести к падению производительности системы. Это, связано с тем, что данные, передаваемые с помощью этих механизмов, копируются из буфера передающего процесса в буфер ядра и затем в буфер принимающего процесса. Механизм разделяемой памяти позволяет избавиться от накладных расходов передачи данных через ядро, предоставляя процессам возможность получить доступ к одной области памяти для обмена данными.

Область разделяемой памяти - это некоторая часть физической памяти, которая используется совместно несколькими процессами. Процесс может присоединить эту область в качестве диапазона виртуальной памяти в свое адресное пространство. Диапазон может быть различным для каждого процесса (см **Рисунок 6.9**). После присоединения процесс получает возможность доступа к этой области как к любому другому участку памяти, больше не нужно использовать системные вызовы для чтения/записи. Поэтому, механизм разделяемой памяти предоставляет процессу максимально быстрый способ доступа к данным. Если процесс записывает данные в ячейки разделяемой памяти, их измененное содержимое сразу же становится видимым остальным процессам, разделяющим между собой эту область.

Рисунок 6.9 Разделяемая область памяти



Таким образом, механизм разделяемой памяти является эффективным средством, позволяющим совместно использовать большие объемы данных без применения копирования и системных вызовов. Основным ограничением механизма является отсутствие средств синхронизации. Если два процесса пытаются изменить одну и ту же разделяемую область памяти ядро системы не может обеспечить последовательность этих операций, а это может привести к смешению записанных данных. Процессы, разделяющие между собой область памяти, должны самостоятельно поддерживать собственный протокол синхронизации. Когда один из процессов записывает данные в разделяемую память, остальные процессы должны ожидать завершения операции. Обычно для этой цели используются семафоры, назначение и число которых определяется конкретным использованием разделяемой памяти. Правда использование семафоров требует вызова одной или нескольких системных функций, а это уменьшает производительность работы с разделяемой памятью.

Разделяемая память POSIX

Для отображения файла в память необходимо сначала его открыть при помощи **open** и затем вызвать **mmap**. Рассмотрим системный вызов **mmap**:

```
paddr=mmap(addr, len, prot, flags, fd, offset);
```

Результатом работы функции становится создание области с образом размером [offset, offset+len] файла fd в адресном пространстве [paddr, paddr+len] процесса. Параметр flag указывает на тип отображения и может принимать значения:

- **MAP_SHARED**,
- **MAP_PRIVATE**.

Переменная **prot** устанавливается как любая комбинация из следующих возможных значений:

- **PROT_READ**,
- **PROT_WRITE**,
- **PROT_EXECUTE**.

Некоторые системы, не поддерживающие привилегии выполнения. Согласованное значение **paddr** выбирается системой. Оно не может быть равным нулю, а образ не вправе накладываться на уже существующие отображения. Вызов **mmap** игнорирует параметр **addr**, если не установлен флаг **MAP_FIXED**. В этом случае значение **paddr**

должно совпадать с **addr**. Если значение **addr** окажется неподходящим (например, не находится в диапазоне корректных адресов процесса), вызов завершится ошибкой.

Системный вызов mmap работает с целыми страницами памяти. Это означает, что параметр **offset** должен быть выровнен по величине страницы. При указании флага **MAP_FIXED** этому требованию должен соответствовать и параметр **addr**. Если значение **len** не совпадает с целым числом страниц, система сама произведет округление до следующего целого числа.

Отображение будет поддерживаться системой до тех пор, пока не будет выполнен вызов:

```
munmap(addr, len);
```

Может быть выполнено замещение адресного диапазона другим файлом при помощи вызова mmap с флагом **MAP_RENAME**. Для изменения признака защиты страницы используется системный вызов:

```
mprotect(addr, len, prot);
```

Разделяемая память System V

Для каждого сегмента разделяемой памяти ядро хранит следующую структуру, определенную в заголовочном файле <sys/shm.h>:

```
struct shmid_ds {
   struct ipc_perm shm_perm; // структура разрешений
size_t shm_segsz; // размер разделяемой памяти
pid_t shm_lpid;
// идентификатор процесса, выполнившего последнюю операцию
pid_t shm_cpid; // идентификатор процесса создателя
shmatt_t shm_nattch; // текущее количество подключений
shmatt_t shm_cnattch; // количество подключений in-core
time_t shm_atime; // Время последнего подключения
time_t shm_dtime; // Время последнего отключения
time_t shm_ctime; // Время последнего изменения данной структуры
};
```

Для создания или для доступа к уже существующей разделяемой памяти используется системный вызов **shmget**:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, int size, int shmflag);
```

Функция возвращает дескриптор разделяемой памяти в случае успеха, и -1 в случае неудачи. Аргумент **size** определяет размер создаваемой области памяти в байтах. Значения аргумента **shmflag** задают права доступа к объекту и специальные флаги **IPC_CREAT** и **IPC_EXCL**. Вызов **shmget** лишь создает или обеспечивает доступ к разделяемой памяти, но не позволяет работать с ней.

Для работы с разделяемой памятью (чтение/запись) необходимо сначала ***присоединить*** (***attach***) ***область*** вызовом **shmat**:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmat(int shmid,char *shmaddr, int shmflag);
```

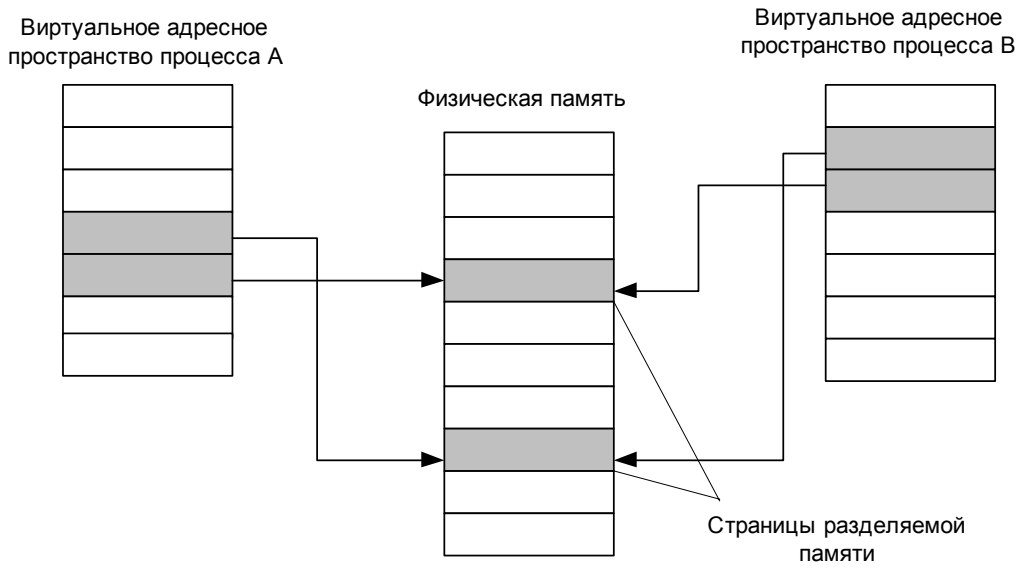
Вызов **shmat** возвращает адрес начала области в адресном пространстве процесса размером size, заданным предшествующим вызовом **shmget**. В этом адресном пространстве взаимодействующие процессы могут размещать требуемые структуры данных для обмена информацией. Правила получения этого адреса следующие:

- Если аргумент **shmaddr** нулевой, то система самостоятельно выбирает адрес.
- Если аргумент **shmaddr** отличен от нуля, значение возвращаемого адреса зависит от наличия флажка **SHM_RND** в аргументе **shmflag**:
  - Если флажок **SHM_RND** не установлен, система присоединяет разделяемую память к указанному **shmaddr** адресу;
  - Если флажок **SHM_RND** установлен, система присоединяет разделяемую память к адресу, полученному округлением в меньшую сторону **shmaddr** до некоторой определенной величины **SHMLBA**.

По умолчанию разделяемая память присоединяется с правами на чтение и запись. Эти права можно изменить, указав флажок **SHM_RDONLY** в аргументе **shmflag**.

Таким образом, несколько процессов могут отображать область разделяемой памяти в различные участки собственного виртуального адресного пространства, как это показано на рисунке (*Рисунок 6.10*).

Рисунок 6.10 Отображение разделяемой памяти



Окончив работу с разделяемой памятью, процесс ***отключает*** (***detach***) ***область*** вызовом **`shmdt`**:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmdt(char *shmaddr);
```

Для полного удаления области процессу необходимо использовать системный вызов **`shmctl`** с **`IPC_RMID`**.

Чему нужно научиться?

Изучить использование разделяемой памяти System V и Posix.

Задание

Уровень 1 (A)

Посмотрите на *"Скелет кода для задания 1 (A)"*, **разберитесь и объясните, что там происходит.**

Уровень 2 (A)

Посмотрите на *"Скелет кода для задания 2 (A)"*, **разберитесь и объясните, что там происходит.**

Уровень 3 (А)

Разработайте наше любимое клиент-серверное приложение с использованием разделяемой памяти:

- Клиент будет считывать полное имя файла из стандартного потока ввода и записывать его в разделяемую память;
- Сервер будет считывать это имя из разделяемой памяти и пытаться открыть файл с этим именем;
  - Если файл успешно откроется, сервер будет передавать его содержимое в разделяемую память. В противном случае он запишет туда сообщение об ошибке.
- Клиент будет считывать данные из разделяемой памяти и записывать их в стандартный поток вывода.
  - Клиент либо выведет содержимое файла, либо сообщение об ошибке.

Скелет кода для задания 1 (А)

shmem.h

```
#define MAXBUFF80
#define PERM0666
/*Структура данных в разделяемой памяти*/
typedef struct mem_msg{
  int segment;
  char buff[MAXBUFF];
} Message;
/*Ожидание начала выполнения клиента*/
static struct sembuf proc_wait[1] = {
  1, -1, 0};
/*Уведомление сервера о том, что клиент начал работу*/
static struct sembuf proc_start[1] = {
  1, 1, 2};
/*Блокировка разделяемой памяти*/
static struct sembuf mem_lock[2] = {
  0, 0, 0,
  0, 1, 0};
/*Освобождение ресурса*/
static struct sembuf mem_unlock[1] = {
  0, -1, 0};
```

Server.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include "shmem.h"
```

```
main()
{
Message*msgptr;
key_tkey;
intshmid, semid;
/*Получим ключ. Один и тот же ключ можно использовать как для семафора, так
и для разделяемой памяти*/
  if(( key = ftok("server", 'A'))<0){
    printf("Невозможно получить ключ\n"): exit(1);}
/*Создадим область разделяемой памяти*/
  if((shmid = shmget(key, sizeof(Message),
           PERM|IPC_CREAT))<0){
    printf("Невозможно создать область\n");
exit(1);}
/*Присоединим ее*/
  if((msgptr = (Message *)shmat(shmid, 0, 0))<0){
    printf("Ошибка присоединения\n"); exit(1);}
/*Создадим группу из двух семафоров:
Первый семафор - для синхронизации работы с разделяемой памятью
Второй семафор - для синхронизации выполнения процессов*/
  if((semid = semget(key, 2, PERM|IPC_CREAT))<0){
    printf("Невозможно создать семафор\n");
exit(1);}
/*Ждем, пока клиент начнет работу и заблокирует разделяемую память*/
  if(semop(semid, &proc_wait[0], 1)<0){
    printf("Невозможно выполнить операцию\n");
exit(1);}
/*Ждем, пока клиент закончит запись в разделяемую память и совободит ее.
После этого заблокируем ее*/
  if(semop(semid, &proc_lock[0], 2)<0){
    printf("Невозможно выполнить операцию\n");
exit(1);}
/*Выведем сообщение на терминал*/
  printf("%s", msgptr->buff);
/*Освободим разделяемую память*/
  if(shmdt(msgptr)<0){
    printf("Ошибка отключения\n"); exit(1);}
/*Всю остальную работу по удалению объектов сделает клиент*/
  exit(0);
}
```

Client.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include "shmem.h"
main()
{
Message*msgptr;
key_tkey;
intshmid, semid;
/*Получим ключ. Один и тот же ключ можно использовать как для семафора, так
и для разделяемой памяти*/
  if(( key = ftok("server", 'A'))<0){
    printf("Невозможно получить ключ\n"): exit(1);}
```

```
/*Получим доступ к разделяемой памяти*/
  if((shmid = shmget(key, sizeof(Message), 0))<0){
    printf("Ошибка доступа\n"); exit(1);}
/*Присоединим ее*/
  if((msgptr = (Message *)shmat(shmid, 0, 0))<0){
    printf("Ошибка присоединения\n"); exit(1);}
/*Получим доступ к семафору*/
  if((semid = semget(key, 2, PERM))<0){
    printf("Ошибка доступа\n"); exit(1);}
/*Заблокируем разделяемую память*/
  if(semop(semid, &proc_lock[0], 2)<0){
    printf("Невозможно выполнить операцию\n");
exit(1);}
/*Уведомим сервер о начале работы*/
  if(semop(semid, &proc_start[0], 1)<0){
    printf("Невозможно выполнить операцию\n");
exit(1);}
/*Запишем в разделяемую память сообщение*/
  sprintf(msgptr->buff, "Good luck!\n");
/*Освободим разделяемую память*/
  if(semop(semid, &proc_unlock[0], 1)<0){
    printf("Невозможно выполнить операцию\n");
exit(1);}
/*Ждем, пока сервер в свою очередь не освободит разделяемую память*/
  if(semop(semid, &proc_lock[0], 2)<0){
    printf("Невозможно выполнить операцию\n");
exit(1);}
/*Отключимся от области*/
  if(shmdt(msgptr)<0){
    printf("Ошибка отключения\n"); exit(1);}
/*Удалим созданные объекты IPC*/
  if(shmctl(shmid, IPC_RMID, 0)<0){
    printf("Невозможно удалить область\n");
    exit(1);}
  if(semctl(semid, 0, IPC_RMID)<0){
    printf("Невозможно удалить семафор\n");
    exit(1);}
  exit(0);
}
```

Скелет кода для задания 2 (А)

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <semaphore.h>

int main(int argc,char **argv)
{
  int fd,i,nloop;
  char *ptr;
  sem_t *mutex;
  nloop=10;
```

```
   if((fd=open("home/tvk/IPC/filedata",O_RDONLY))<0)
   {
     printf("Can not open file name\n");
exit(1);
   }
   printf("File is opened!\n");
   ptr=mmap(NULL,sizeof(char),PROT_READ|PROT_WRITE,
MAP_SHARED,fd,0);
   close(fd);
   mutex=sem_open(px_ipc_name("mysem"),O_CREAT|O_EXCL,0);
   sem_unlink(px_ipc_name("mysem"));
   setbuf(stdout,NULL);
   if(fork()==0) {
     for(i=0;i<nloop;i++)
     {
       sem_wait(mutex);
       printf("Child: %c\n",(*ptr)++);
       sem_post(mutex);
     }
     exit(0);
       }
   for(i=0;i<nloop;i++)
   {
     sem_wait(mutex);
     printf("Parent: %c\n",(*ptr)++);
     sem_post(mutex);
   }

exit(0);
}
```

Глава 7

Файловая система (FS)

Практически всем приложениям нужно где-то хранить информацию. Можно хранить данные в оперативной памяти но:

- оперативная память теряет свое содержимое после отключения питания;
- объем данных может превышать ее возможности;
- кроме того, информацию нужно хранить в виде, независимом от процессов.

Поэтому данные хранят на внешних носителях (обычно это диски) в файлах. Вначале каждая прикладная программа сама решала проблему организации хранения информации, а затем появились централизованные системы управления внешней памятью.

Введем основные определения.

Файл - именованный блок данных.

Файловая система - логическая структура, организованная на базе последовательно адресуемого адресного пространства на внешнем устройстве и обеспечивающая уникальное именование файлов и доступ к данным файлов по их именам. Для файловой системы определено множество операций.

Система управления файлами - это часть ОС, назначение которой состоит в том, чтобы организовать эффективную работу с данными, хранящимися во внешней памяти и обеспечить пользователю удобный интерфейс при работе с этими данными.

При работе с файлами ОС хранит в оперативной памяти некоторую информацию о файлах, например, расположение файла на внешнем носителе, права доступа к файлу и т.д. В этом случае говорят о системной таблице индексных дескрипторов файлов.

Стандартом логической организации хранилища файлов является иерархическая структура, называемая ***деревом каталогов*** (***директорий***).

Каталог (***директория***) - специальный тип объекта файловой системы, который не хранит данные, но содержит другие объекты файловой системы, например, файлы и подкаталоги.

Итак, файловая система включает:

- совокупность всех файлов на диске;
- наборы структур данных, используемых для управления файлами, такие, например, как каталоги, дескрипторы файлов, таблицы распределения свободного и занятого пространства на диске и т.д.;
- комплекс системных программных средств, реализующих управление файлами, в частности: создание, уничтожение, чтение, запись, именование, поиск и другие операции над файлами.

Перечислим основные функции файловой системы:

- идентификация файлов. Связывание имени файла с выделенным ему пространством внешней памяти;
- распределение внешней памяти между файлами;
- поддержка операций управления объектами файловой системы;
- обеспечение надежности и отказоустойчивости. Стоимость информации может во много раз превышать стоимость компьютера;
- обеспечение защиты от несанкционированного доступа;
- обеспечение совместного доступа к файлам, не требуя от пользователя специальных усилий по обеспечению синхронизации доступа;
- обеспечение высокой производительности.

Файлы и каталоги

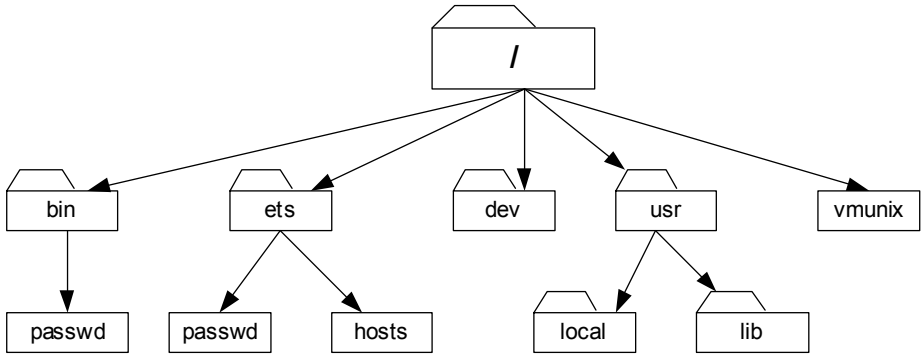
Логически файл представляет собой место для хранения данных. Пользователь может создать файл, записать туда информацию, а затем осуществлять доступ к файлу для получения этой информации. Доступ к данным может быть как последовательным, так и произвольным. Ядро системы поддерживает различные операции управления файлами, позволяющие:

- называть файлы;
- использовать файлы;
- управлять доступом к файлам.

Ядро не интерпретирует содержание или структуру файла, для него это просто последовательность байтов, к которой оно предоставляет побайтовый доступ.

С точки зрения пользователя файлы в системе UNIX организованы в виде иерархического древовидного пространства имен (см. *Рисунок 7.1*). Дерево состоит из файлов и каталогов. Каталог содержит информацию об именах файлов и других каталогов, находящихся в нем. Каждое имя файла или каталога может содержать любые символы ASCII, кроме / и **NULL**. Файловая система может ограничивать длину имен файлов. ***Корневой каталог*** обозначается /. Имена файлов должны отличаться друг от друга только в пределах одного каталога. Для уникальной идентификации файла необходимо указывать ***полное имя файла*** (***pathname***). Полное имя включает все каталоги и подкаталоги, начиная от корневого каталога до файла, разделенные между собой символом /.

Рисунок 7.1 Файлы и каталоги



В UNIX текущий рабочий каталог для каждого процесса, хранится как часть информации о нем в таблице процессов. Это позволяет пользователям оперировать с относительными именами файлов, интерпретируемыми в зависимости от текущего каталога. Для этого применяются два обозначения:

- **.** (одна точка) обозначает текущий каталог;
- **..** (две точки) обозначают родительский каталог.

Таблица индексных дескрипторов

Ядро хранит информацию о каждом файле в специальной структуре на диске, называемой ***индексным дескриптором файла*** (***index node***, ***inode***). Формат и содержимое индексных дескрипторов отличается в различных файловых системах. При создании файловой системы формируется ***таблица индексных дескрипторов***. Если файл открыт или каталог является текущим, ядро сохраняет данные о них из дисковой копии индексного дескриптора в структуре данных, размещенной в памяти, также называемой ***индексным дескриптором***. Будем использовать термин ***индексный дескриптор на диске*** (***on disk inode***) для указания на структуру данных, размещенных на диске (***dinode***), а термин ***индексный дескриптор в памяти*** (***in core inode***) для обозначения структуры, размещаемой в оперативной памяти (***inode***).

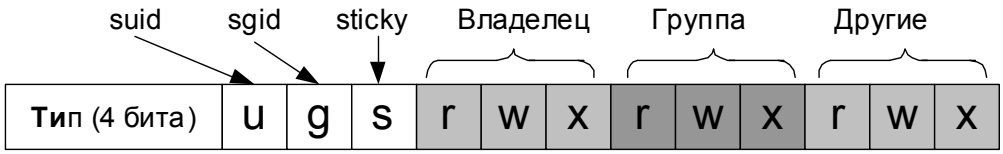
Поля структуры ***dinode*** для файловой системы System V (s5fs) представлены в таблице (см. *Таблица 7.1*).

Таблица 7.1 Поля структуры ***dinode***

Поле	*Размер в байтах*	*Описание*
di_mode	2	***Тип файла и полномочия доступа***
di_nlinks	2	***Количество (счетчик) ссылок***
di_uid	2	Идентификатор пользователя - владельца
di_gid	2	Идентификатор группы пользователя - владельца
di_size	4	Размер файла в байтах
di_addr	39	***Массив адресов блоков***
di_gen	1	Генерируемый номер (инкрементируется каждый раз при запросе индексного дескриптора для нового файла)
di_atime	4	Время последнего доступа
di_mtime	4	Время последней модификации
di_ctime	4	Время последнего изменения индексного дескриптора (кроме изменений полей di_atime di_mtime)

Рассмотрим поле di_mode (см. *Рисунок 7.2*).

Рисунок 7.2 Структура поля di_mode



Первые четыре бита определяют тип файла. Система UNIX умеет распознавать несколько типов файлов, в том числе:

- обычные файлы;
- каталоги;
- файлы FIFO;
- жесткие ссылки (hard links);
- символические ссылки (symbolic links);
- специализированные файлы, представляющие блочные или символьные устройства.

Каталог - это специальный файл, который содержит список файлов и подкаталогов. В нем находятся записи фиксированного размера (по 16 байт). В первых двух байтах размещается номер индексного дескриптора, а следующие 14 байт отведены для имени файла (см. *Рисунок 7.3*). Используя каталог, для каждого имени можно определить номер соответствующего ему индексного дескриптора.

Индексный дескриптор не содержит ни имени файла, ни указателя на это имя. Такая архитектура позволяет одному файлу иметь несколько имен в файловой системею

Жесткая ссылка - это не тип файла, а его дополнительное имя (псевдоним). Жесткие ссылки позволяют нескольким элементам каталогов разделять один и тот же индексный дескриптор (см. *Рисунок 7.4*). Для создания жесткой ссылки используется команда **`ln`**.

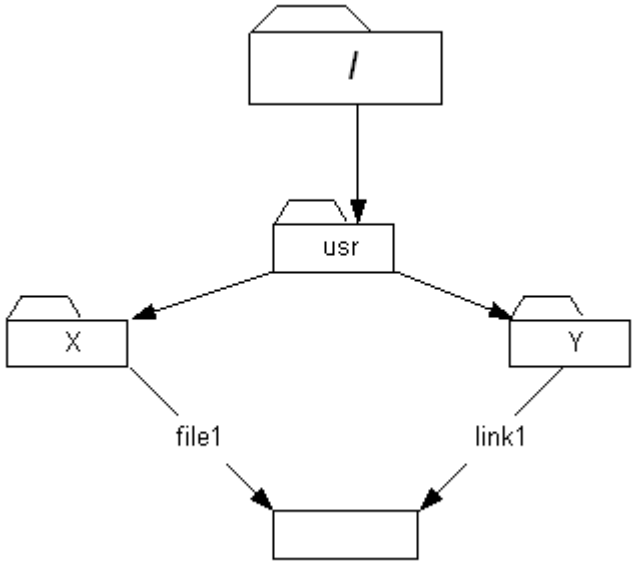
Таким образом, все файловые ссылки эквивалентны и являются просто различными именами одного и того же файла.

Доступ к файлу можно осуществлять по любой ссылке, и не определить, какая из ссылок является изначальной.

Рисунок 7.3 Каталог

73	.
38	..
9	file1
0	deletedfile
110	subdirectory
65	archana

Рисунок 7.4 Жесткие ссылки



Каждый раз при создании жесткой ссылки на файл ***счетчик ссылок*** увеличивается на 1, а при удалении уменьшается на 1. Сам файл будет удален только в том случае, если счетчик ссылок станет равным нулю.

Современные варианты UNIX поддерживают еще ***символические ссылки***. Для создания символической ссылки служит вызов **`ln -s`**. ***Символическая ссылка*** - это специальный файл, указывающий на

другой файл. Для идентификации файла как символической ссылки используются ее атрибуты. Внутри такого файла содержится путь к связываемому файлу. Многие системы позволяют сохранять короткие имена путей. Полное имя символической ссылки может быть как абсолютным, так и относительным. Процедуры преобразования путей опознают символические ссылки и преобразуют их для получения имени файла. Если используется относительное имя, то оно интерпретируется в зависимости от каталога, в котором находится ссылка.

Теперь перейдем к рассмотрению следующих трех битов:

- бит смены идентификатора пользователя - **SetUID**;
- бит смены идентификатора группы - **SetGID**;
- бит фиксации("липучка") - **sticky**.

Первые два бита относятся к выполняемым файлам. Если пользователь выполняет файл, для которого был установлен бит **SetUID**, ядро изменит действительный идентификатор пользователя на идентификатор владельца файла. Бит **SetGID** используется для аналогичного изменения действительного группового идентификатора.

Флаг навязчивости **sticky** нужен также для выполняемых файлов и используется для запроса к ядру системы на сохранение "образа" программы в области свопинга после завершения ее выполнения.

С каждым файлом ассоциируются следующие права доступа:

- право на чтение (**read**);
- право на запись (**write**);
- право на выполнение (**execute**).

Кроме того, все пользователи делятся также на три категории:

- владелец файла (**owner**);
- пользователи, которые входят с владельцем в одну группу (**group**);
- все остальные (**others**).

Следовательно, все права на файл могут быть заданы с помощью 9 битов.

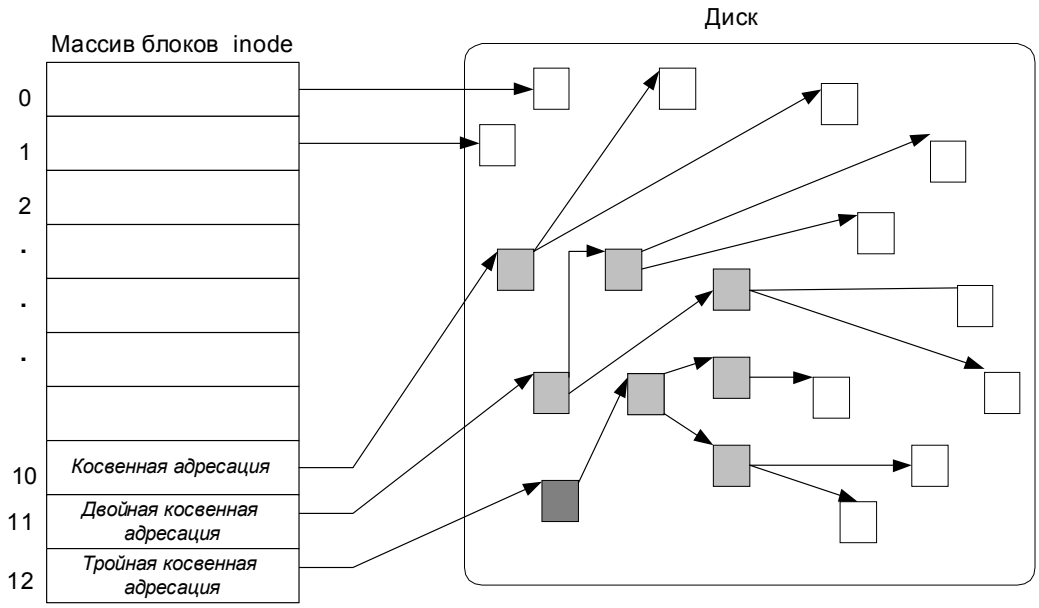
Права на каталоги задаются следующим образом:

- ***право на запись*** в каталог позволяет создавать или удалять файлы в нем;
- ***право на выполнение*** позволяет получить доступ к файлам каталога.

Механизм прав доступа достаточно примитивен, поэтому большинство производителей систем UNIX предлагают дополнительные средства защиты, такие как ***списки контроля доступа*** (***ACL***).

Теперь рассмотрим ***массив адресов блоков***. В ОС UNIX список блоков хранится прямо в индексном дескрипторе, кроме того, используются дополнительные блоки для файлов больших размеров. Этот метод весьма эффективен для маленьких файлов, но недостаточно гибок для больших файлов. Схема размещения представлена на рисунке (*Рисунок 7.5*).

Рисунок 7.5 Массив адресов блоков в индексном дескрипторе



Поле **`di_addr`**, занимающее 39 байт, содержит 13-ти элементный массив, каждый элемент которого хранит в себе 3-байтовый номер физического блока. Элементы от 0 до 9 содержат номер 0-9 блоков данных файла. Таким образом, для файла, умещающегося в 10 и менее блоков, вся адресная информация хранится в индексном дескрипторе. Элемент 10 представляет собой номер блока косвенной адресации с содержимым в виде массива номеров блоков. Элемент 11 указывает на блок двойной косвенной связи, содержащий номера других "косвенных" блоков. И последний, 12-й элемент, указывает на "трижды косвенный" блок с номерами "дважды косвенных" блоков.

В этой схеме при размере блока в 1024 байт первые 10 блоков можно адресовать непосредственно, еще 256 блоков через один блок косвенной связи, еще 65536 (256x256) блоков через двойной косвенный блок и еще 16777216 (256x256x256) блоков - через блок тройной косвенной адресации.

Работа с файлами

В UNIX существует два основных интерфейса для файлового ввода/вывода:

- ***интерфейс системных вызовов***, представляющий системные функции, непосредственно взаимодействующие с ядром ОС. Можно это сформулировать по-другому - ядро ОС UNIX позволяет пользовательским процессам взаимодействовать с файловой системой через определенный интерфейс системных вызовов. Этот интерфейс определяет представление файловой системы с точки зрения пользователя, семантику и действия всех относящихся к ней системных вызовов. Пользователь оперирует такими абстракциями, как ***файлы***, ***каталоги***, ***дескрипторы файлов*** и ***файловые системы***;
- ***стандартная библиотека ввода/вывода***, представляющая функции буферизованного ввода/вывода. Этот интерфейс является надстройкой над интерфейсом системных вызовов и представляет более простой и удобный способ работы с файлами. Функции более высокого уровня, представляемые стандартной библиотекой ввода/вывода, используют системные вызовы.

Далее будут рассмотрены основные системные вызовы для работы с файловой системой (**`open`**, **`read`**, **`write`**, **`lseek`**, **`close`**, **`creat`** и т.д.) и следующие структуры данных ядра:

- ***таблицу файлов***, в которой каждая запись связана с одним из открытых в системе файлов;
- ***таблицу пользовательских дескрипторов файлов***, в которой каждая запись связана с файловым дескриптором, известным процессу;
- ***таблицу монтирования***, в которой содержится информация по каждой активной файловой системе.

Дескрипторы файлов

Введем понятие ***файлового дескриптора***. Перед использованием файла процесс должен сначала открыть его для чтения или записи, используя системный вызов **`open`**. В этом случае ядро будет использовать следующий алгоритм:

- получить по имени файла значение индексного дескриптора;
- если файл не существует или к нему не разрешен доступ, будет возвращен код ошибки;
- выделить для индекса запись в ***таблице файлов***, инициализировать счетчик и смещение;
- выделить запись в ***таблице пользовательских дескрипторов*** и установить указатель на запись в ***таблице файлов***;
- возвратить пользовательский дескриптор файла.

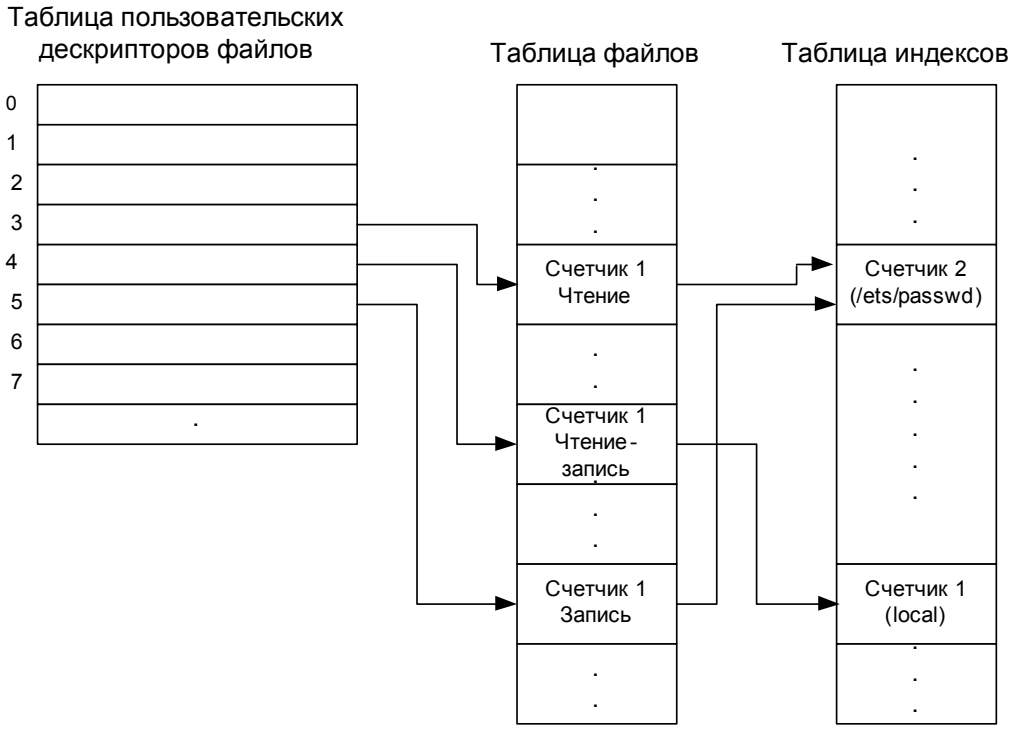
Запись таблицы файлов содержит указатель на индексный дескриптор открытого файла и поле, в котором хранится смещение в байтах от начала файла до места, откуда предполагается начинать выполнение последующих операций чтения или записи. Ядро сбрасывает это смещение в 0 во время открытия файла, имея в виду, что исходная операция чтения или записи по умолчанию будет производиться с начала файла. С другой стороны, процесс может открыть файл в режиме записи в конец, в этом случае ядро устанавливает значение смещения, равное размеру файла. Ядро выделяет запись в таблице пользовательских дескрипторов файлов в адресном пространстве, выделенном процессу, и запоминает указатель на эту запись. Указателем выступает дескриптор файла, возвращаемый пользователю. Запись в таблице пользовательских файлов указывает на запись в глобальной таблице файлов.

Предположим, что процесс, открывая файл **`/etc/passwd`** дважды, один раз только для чтения и один раз только для записи, и однажды файл **`local`** для чтения и для записи:

```
fd1=open("/etc/passwd",O_RDONLY);
fd2=open("local",O_RDWR);
fd3=open("/etc/passwd",O_WRONLY);
```

На Рисунке (см. *Рисунок 7.6*) показана взаимосвязь между таблицей индексов, таблицей файлов и таблицей пользовательских дескрипторов файла.

Рисунок 7.6 взаимосвязь между таблицами индексных дескрипторов, файлов и пользовательских дескрипторов файлов



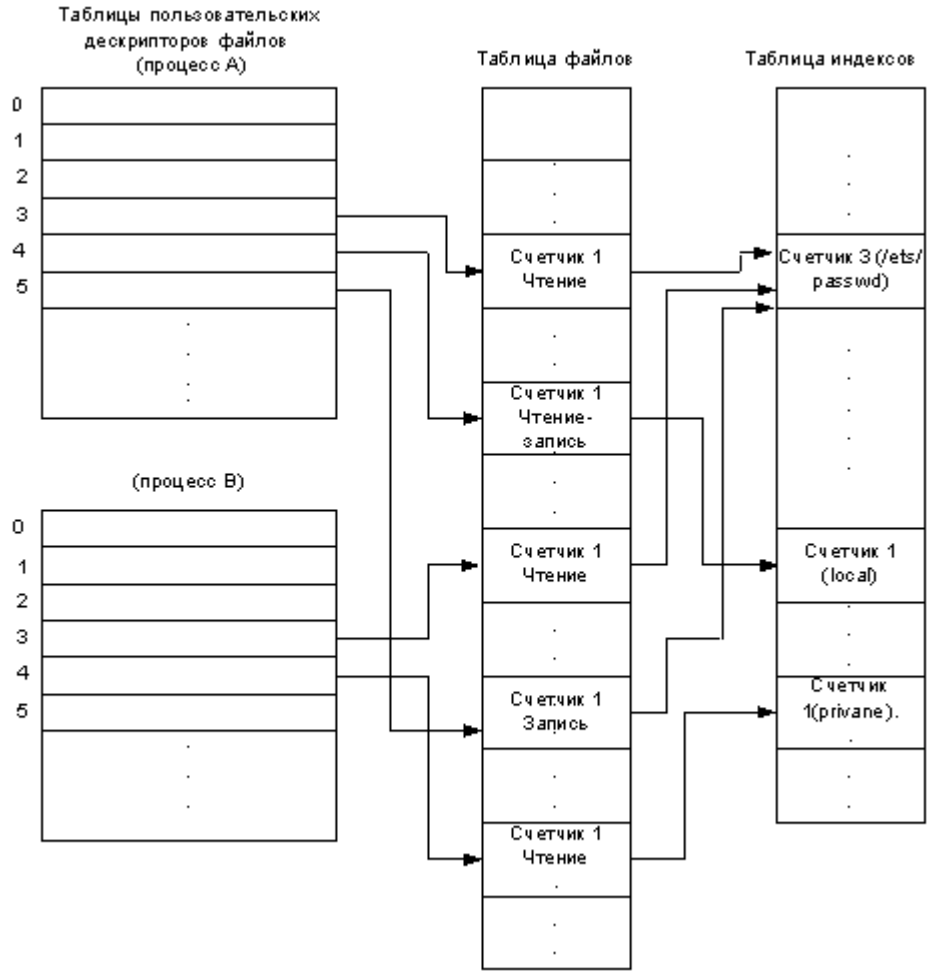
Каждый вызов функции **open** возвращает процессу дескриптор файла, а соответствующая запись в таблице пользовательских дескрипторов файла указывает на уникальную запись в таблице файлов ядра, пусть даже один и тот же файл (**/etc/passwd**) открывается дважды. Записи в таблице файлов для всех экземпляров одного и того же открытого файла указывают на одну запись в таблице индексов, хранящихся в памяти.

Процесс может обращаться к файлу "**/etc/passwd**" с чтением или записью, но только через дескрипторы файла, имеющие значения 3 и 5. Ядро запоминает разрешение на чтение или запись в файл в строке таблицы файлов, выделенной во время выполнения функции open. Предположим, что второй процесс выполняет следующий набор операторов:

```
fd1=open("/etc/passwd",O_RDONLY);
fd2=open("private",O_RDONLY);
```

На Рисунке (см. *Рисунок 7.7*) показана взаимосвязь между соответствующими структурами данных, когда оба процесса (и больше никто) имеют открытые файлы.

Рисунок 7.7 Взаимосвязь таблиц (два процесса)



Снова результатом каждого вызова функции **open** является выделение уникальной точки входа в таблице пользовательских дескрипторов файла и в таблице файлов ядра, и ядро хранит не более одной записи на каждый файл в таблице индексов, размещенных в памяти.

Первые три пользовательских дескриптора (0, 1 и 2) именуются дескрипторами файлов: стандартного ввода, стандартного вывода и стандартного файла ошибок. Процессы в системе UNIX по договоренности используют дескриптор файла стандартного ввода при чтении вводимой информации, дескриптор файла стандартного вывода при записи выводимой информации и дескриптор стандартного файла ошибок для записи сообщений об ошибках.

Двойное открытие файла

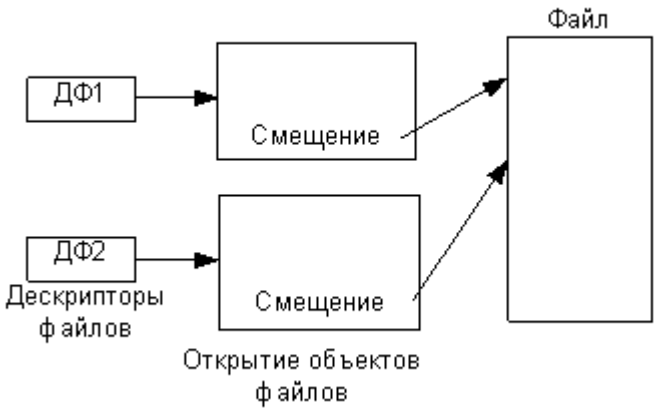
Итак, вызов **`open`** возвращает дескриптор открываемого файла. Пользователь может открывать один и тот же файл неоднократно, его могут одновременно открывать и сразу несколько пользователей. В последнем случае ядро системы создаст новые объекты открытого файла и дескрипторы.

Дескриптор файла относится к определенному процессу. Один и тот же дескриптор для двух процессов может ссылается на разные файлы. Процесс использует дескрипторы для передачи ввода-вывода системным вызовам, таким как **`read`** или **`write`**. Ядру дескрипторы нужны для быстрого нахождения открытого файла, а также других структур данных, ассоциируемых с этим файлом. Это дает возможность ядру производить различные действия, такие как анализ полного имени или управление доступом, на стадии вызова функции **`open`**, а не при проведении каждой операции ввода-вывода. Открытые файлы можно закрыть либо при помощи системного вызова **`close`**, либо это произойдет автоматически при завершении работы процесса.

Каждый дескриптор связан с независимым сеансом работы с файлом. Ассоциируемый с этим дескриптором объект открытого файла содержит контекст сеанса. Контекст включает в себя режимы, при которых было произведено открытие файла, а также указатель смещения, показывающий точку начала следующей операции чтения или записи. В системе UNIX по умолчанию доступ к файлам осуществляется последовательно. После открытия файла пользователем ядро производит инициализацию указателя смещения в значение "ноль". В дальнейшем каждая операция чтения или записи увеличит значение указателя на количество переданных данных.

Если два процесса (см. *Рисунок 7.8*) открывают один и тот же файл, то операция чтения или записи одним из этих процессов приведет к сдвигу только собственного указателя смещения. Значение второго указателя при этом не изменяется. Такой подход обеспечивает прозрачное совместное использование файла несколькими процессами одновременно. В большинстве случаев доступ к одному и тому же файлу несколькими процессами влечет необходимость применения для его блокировки технологии синхронизации.

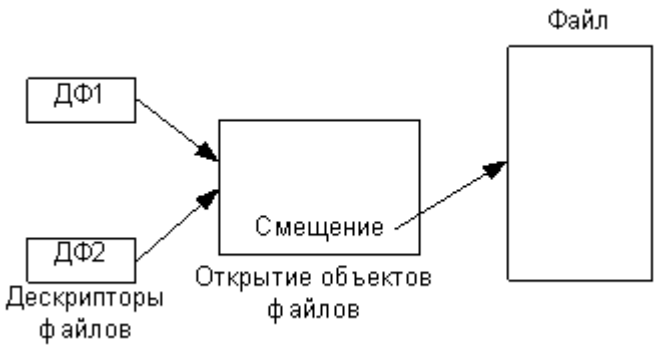
Рисунок 7.8 Файловые дескрипторы при двойном открытии файла



Дублирование файловых дескрипторов

Процесс может дублировать дескрипторы при помощи системных вызовов **dup** или **dup2**. Эти функции создают новый дескриптор, указывающий на один и тот же объект открытого файла и, следовательно, разделяющий один и тот же сеанс (см. *Рисунок 7.9*).

Рисунок 7.9 Продублированный дескриптор



Системный вызов **fork** производит создание копий всех файловых дескрипторов родительского процесса и передает их потомку. После возврата из **fork** предок и потомок разделяют между собой одинаковый набор открытых файлов. При таком варианте совместного использования, так как два дескриптора разделяют между собой один и тот же сеанс работы, они оба видят файл одинаково и используют тождественный указатель смещения. Операция увеличения указателя смещения, произведенная над одним из дескрипторов, приведет к тому, что эти изменения будут видны и в остальных.

Процесс может передавать дескриптор файла любому другому процессу, при этом достигается эффект передачи ссылки на объект открытого файла. Ядро производит копирование дескриптора в первую свободную ячейку таблицы дескрипторов получателя. В результате оба дескриптора разделяют между собой один и тот же объект и, следовательно, обладают единым указателем смещения. Обычно процесс, посылающий дескриптор, закрывает его после завершения операции отправки. Это не приводит к закрытию файла даже в том случае, если адресат так и не получил дескриптор, поскольку ядро системы удерживает второй дескриптор во время проведения передачи.

Файловый ввод/вывод

Система UNIX позволяет осуществлять как произвольный, так и последовательный доступ к файлам. По умолчанию применяется последовательный доступ. Ядро поддерживает указатель смещения файла, который при первом открытии этого файла инициализируется в ноль. Он указывает на текущую позицию в файле, с которой будет производиться следующая операция ввода-вывода. Каждый раз, когда процесс считывает или записывает данные в файл, ядро сдвигает указатель на величину переданного объема данных. Произвольный доступ к файлу достигается с помощью системного вызова **`lseek`**, которому передается необходимое значение указателя смещения. Последующий вызов функции **`read`** или **`write`** приведет к передаче данных, начиная с указанной позиции.

Ядро считывает данные из файла, ассоциированного с файловым дескриптором, начиная от смещения, сохраненного в объекте открытого файла. Возможно возникновение ситуации, когда количество считываемых байтов меньше, чем величина **`count`**. Это может произойти при достижении конца файла или в случае отсутствия доступных данных при обращении к файлам FIFO или устройствам.

Вызов **`read`** возвращает количество переданных байтов (**`nread`**). Эта функция также производит смещение указателя на **`nread`** байтов, поэтому следующий вызов **`read`** или **`write`** начнет передачу данных с точки завершения действий предыдущей функции.

Хотя ядро позволяет нескольким процессам производить открытие и совместное использование файла, на самом деле эти операции последовательны. Например, если двум процессам необходимо одновременно произвести запись в один и тот же файл, ядро системы произведет вторую операцию записи только после завершения первой. Такой подход позволяет каждой операции работать с файлами, находящимися в непротиворечивом состоянии.

Файловые системы

Файловая система располагается на одном логическом диске или в разделе. Каждый логический диск может содержать в себе лишь одну файловую систему. Всякая файловая система является независимой и содержит собственный корневой каталог, подкаталоги, файлы, а также ассоциированные с ними данные и метаданные. Файловое дерево, видимое пользователем, формируется из одной или нескольких файловых систем. В начале раздела находится загрузочная область, содержащая код, необходимый для начальной загрузки и инициализации ОС (см.*Рисунок 7.10*).

Рисунок 7.10 Структура файловой системы

В принципе, для хранения подобной информации достаточно одного раздела диска, однако каждый имеющийся на диске дополнительный раздел должен по возможности иметь такую область (пустую). После области загрузки на диске располагается суперблок, который содержит атрибуты и метаданные файловой системы. Далее размещается ***список индексных дескрипторов***, являющийся одномерным массивом индексных дескрипторов. Такой дескриптор имеется у каждого файла. Каждый дескриптор идентифицируется при помощи номера, эквивалентного индексу в списке дескрипторов. Размер дескриптора составляет 64 байт. В одном дисковом блоке могут быть размещены несколько дескрипторов. Начальные смещения для суперблока и списка индексных дескрипторов одинаковы во всех разделах системы. Следовательно, номер дескриптора может быть легко преобразован в номер блока и смещение от начала этого блока. Список индексных дескрипторов имеет фиксированный размер (устанавливаемый при создании файловой системы на разделе) который ограничивает максимально допустимое число файлов в этом разделе. После таблицы дескрипторов на диске следует область данных. В ней расположены блоки данных для хранения файлов и каталогов, а также блоки косвенной адресации.

Суперблок

Суперблок содержит метаданные применительно к файловой системе. В каждой файловой системе имеется один суперблок, располагающийся в начале диска. Ядро считает данные из суепрблока при монтировании файловой системы и хранит их в памяти до тех пор, пока система не будет размонтирована. Суперблок содержит следующую информацию:

- размер файловой системы в блоках;
- размер списка индексных дескрипторов в блоках;
- количество свободных блоков и индексных дескрипторов;
- список свободных блоков;
- список сводных индексных дескрипторов.

Логические диски

Логический диск - это элемент, используемый для хранения информации, видимый ядром системы как линейная последовательность блоков фиксированного размера, доступных в произвольном порядке. Драйвер дискового устройства производит отображение блоков на физические средства хранения. Для создания файловой системы на диске в UNIX используется **`mkfs`**. Каждая файловая система занимает один логический диск. Логический диск может содержать только одну файловую систему. На некоторых логических дисках файловая система может отсутствовать, например, если диск используется подсистемой памяти для свопинга.

Поддержка логических дисков позволяет размечать физический объем устройства различными способами. В простейшем случае каждый логический диск отображает все физическое пространство диска полностью. Обычно диск разделяется на некоторое количество непрерывных разделов, каждый из которых является логическим устройством. В ранних системах UNIX встречался только такой способ разбиения дисков. В результате слово "раздел" и сейчас часто применяется для описания физического пространства файловой системы.

Специальные файлы

Одной из отличительных черт системы UNIX является распространения понятия файла на все объекты, относящиеся к вводу-выводу, в том числе каталоги символических ссылок, аппаратные устройства (диски, терминалы, и принтеры), псевдоустройства

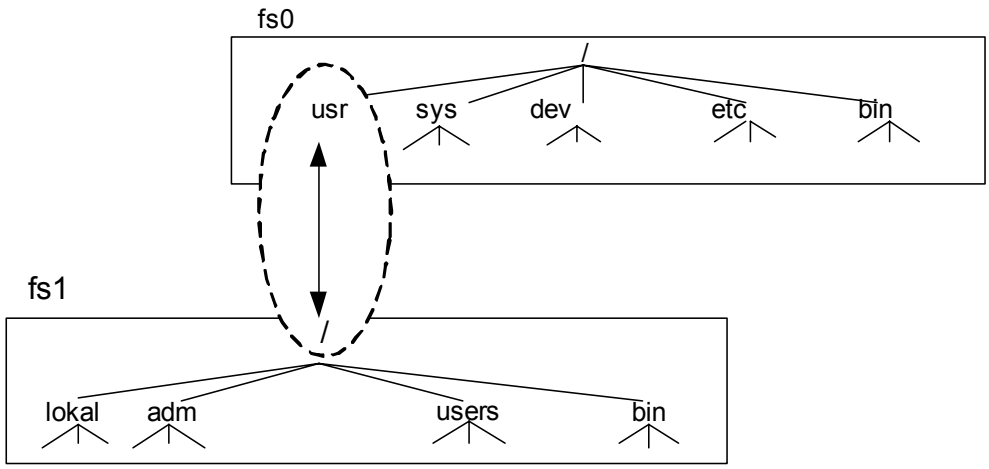
(системная память) а также элементы коммуникаций (каналы и сокеты). Доступ к каждому из перечисленных объектов производится путем обращения к дескриптору файла. Действия над специальными файлами осуществляются при помощи тех же системных вызовов, что и для работы с обычными файлами. Например, пользователь может послать данные на принтер простым открытием файла, ассоциированным с указанным принтером, предварительно сохранив информацию в этом файле.

Монтирование

Файловая иерархия системы UNIX выглядит как монолитная структура, но в реальности она представляет собой композицию нескольких отдельных деревьев, каждое из которых является отдельной, полной файловой системой. Одна из файловых систем настраивается как корневая файловая система, а ее корневой каталог становится системным корневым каталогом. Остальные файловые системы присоединяются к существующей структуре при помощи монтирования каждой новой файловой системы в каталог, находящийся в существующем дереве. После монтирования корневой каталог системы "покрывается" каталогом, на которой система была смонтирована. Любой доступ к монтировочному каталогу преобразуется в доступ к корневому каталогу смонтированной файловой системы. Она остается видимой до тех пор, пока не будет произведена операция размонтирования.

На рисунке (*Рисунок 7.11*) показана иерархия файлов, получаемая при композиции двух файловых систем.

Рисунок 7.11 Монтирование файловой системы



В примере система **fs0** установлена как корневая файловая система машины, а **fs1** смонтирована в каталоге **/usr** системы **fs0**.

Каталог **/usr** называется ***монтировочным*** или ***точкой монтирования***. Все попытки доступа к нему приведут к обращению к корневому каталогу монтированной на нем файловой системы.

Архитектура vnode/vfs

Интерфейс vnode/vfs разработан компанией Sun Microsystems как базовая структура, поддерживающая различные файловые системы. Описываемая технология получила широкое распространение и стала частью System V Unix в SVR4.

При разработке архитектуры vnode/vfs выдвигались следующие основные требования:

- Необходимость одновременной поддержки несколько различных типов файловых систем, в том числе для UNIX (s5fs или ufs) и других ОС (например, DOS, A/UX и т. д.).
- Поддержка содержания в разных разделах одного диска различных типов файловых систем. После монтирования разделы должны быть представлены для пользователя в обычном виде, как однородная файловая система. Пользователь при этом должен видеть дерево файлов полностью и не беспокоиться насчет различий в представлении на дисках каждого поддерева системы.
- Система должна обеспечивать разделение файлов по сети. При этом удаленная файловая система обязана быть доступна пользователю точно так же, как и система на локальной машине.
- Производители могут создавать собственные типы файловых систем и добавлять их в ядро в качестве модулей.

ЛАБОРАТОРНАЯ РАБОТА 7

ФАЙЛОВАЯ СИСТЕМА

Теория

Open

Перед использованием файла процесс должен сначала открыть его для чтения или записи. Системный вызов **open** имеет следующий синтаксис:

```
#include <fcntl.h>

int fd=open(const char *path, int oflag, mode_t mode);
```

где **path**- это абсолютное или относительное имя файла, а **mode** указывает на права, ассоциируемые с файлом при его создании. Флаги, передаваемые при помощи **oflag**, указывают на необходимость открытия файла после его создания на:

- чтение (**read**) - **O_RDONLY**;
- запись (**write**) - **O_WONLY**;
- чтение/запись (**read/write**) - **O_RDWR**;
- добавление (**append**) - **O_APPEND**;
- и т. д.

В случае неудачи **open** возвратит -1 (errno будет содержать код ошибки), а в случае удачи будет возвращен дескриптор файла.

Каждый процесс имеет маску создания файлов, определенную по умолчанию, являющуюся битовой маской прав, не предоставляемых создаваемым файлам. Когда пользователь указывает параметр **mode** для функции **open** или **creat**, пользователь обнуляет разряды, указанные в маске по умолчанию. Для изменения маски, принятой по умолчанию, используется системный вызов **chmod**.

Creat

Для создания файла также применяется системный вызов **creat**, который эквивалентен вызову **open** с флагами **O_WRONLY**, **O_CREAT** и **O_TRUNC**.

```
#include <fcntl.h>

int creat(const char *path, mode_t mode);
```

как и в случае **open**, **path**- это абсолютное или относительное имя файла, а **mode** устанавливаемые права доступа к файлу.

Close

Функция **close** разрывает связь между файловым дескриптором и открытым файлом, созданную функциями **creat**, **open**, **dup** и **fcnt**l.

```
#include <fcntl.h>

int close(int fildes);
```

В случае неудачи close возвратит -1 (**errno** будет содержать код ошибки), а в случае удачи будет возвращен 0.

Функция **exit** автоматически закрывает открытые файлы.

Dup

Функция **dup** используется для дублирования существующего файлового дескриптора:

```
int dup(int fildes);
```

Файловый дескриптор должен быть предварительно получен с помощью **creat**, **open** и **dup**. В случае успешного завершения функция возвращает новый файловый дескриптор, свойства которого идентичны свойствам дескриптора **fildes**.

Lseek

С файловым дескриптором связан файловый указатель, определяющий текущее смещение в файле, начиная с которого будет произведена последующая операция чтения или записи. С помощью **lseek** можно установит файловый указатель на любое место файла;

```
#include <unistd.h>

off_t lseek(int fildes, off_t offset, int whence);
```

Интерпретация аргумента **offset** зависит от аргумента **whence**, который может принимать следующие значения:

- **SEEK_CUR**. Указатель смещается на **offset** от текущего положения;
- **SEEK_END**. Указатель смещается на **offset** байт от конца файла;
- **SEEK_SET**. Указатель устанавливается равным **offset**.

Read и Write

Вызовы **read** и **write** имеют одинаковую семантику, поэтому приведем пример только одной из этих функций:

```
#include <unistd.h>

ssize_t nread=read(int fd, void *buf, size_t count);
```

где **fd** - это дескриптор файла, **buf** - указатель на буфер в пользовательском адресном пространстве, в который должно производиться чтение данных, а параметр **count** - количество считываемых байтов.

Возможно возникновение ситуации, когда количество считываемых байтов меньше, чем величина **count**. Это может произойти при достижении конца файла или в случае отсутствия доступных данных при обращении к файлам FIFO или устройствам.

Ответственность за емкость буфера **buf**, достаточную для помещения count байтов данных, несет пользователь. Вызов **read** возвращает количество переданных байтов (**nread**). Эта функция также производит смещение указателя на nread байтов, поэтому следующий вызов **read** или **write** начнет передачу данных с точки завершения действий предыдущей функции.

Файл может быть открыт в режиме добавления, для чего системному вызову open необходимо передать флаг **O_APPEND**. В этом случае перед вызовом write для указанного дескриптора ядро установит указатель смещения на конец файла. Если пользователь открывает файл в режиме добавления, то это не повлияет на операции с этим файлом, производимые через другие дескрипторы.

Fcntl

После открытия файла и получения файлового дескриптора процесс может производить различные файловые операции. Функция **fcntl** позволяет процессу выполнить ряд действий с файлом, используя его дескриптор:

```
#include <fcntl.h>

int fcntl(int fildes, int cmd,…);
```

Функция выполняет действие **cmd** с файлом, а третий аргумент зависит от конкретного действия:

- **F_GETLK**. Проверить существование блокирования записи файла. Блокирование описывается структурой **flock**, указатель на которую передается в качестве третьего аргумента;

- **F_SETLK**. Установить блокирования записи файла. Структура **flock** описывает блокирование и указатель на нее передается в качестве третьего аргумента;
- **F_SETLKW**. Аналогично предыдущему, но при невозможности блокирования по причине уже существующих блокировок, процесс переходит в состояние сна, ожидая пока последние освободятся.

Чему нужно научиться?

Изучить основные системные вызовы для работы с файлами.

Задание

Уровень 1, 2, 3 (А)

Необходимо чтобы два процесса заносили очередной элемент в связанный список, который хранится в файле.

Приложение

Подключение файловых систем (монтирование)

Пользователи видят дерево каталогов и файлов как единую структуру. На самом деле различные каталоги этого дерева могут соответствовать различным разделам диска, находиться на различных дисках и даже на других компьютерах. Раздел диска присоединяется к дереву в каталоге, называемом ***точкой подключения*** или ***монтирования*** (***mount point***). Точка подключения и все расположенные ниже ее каталоги образуют файловую систему. Для монтирования файловой системы в дерево каталогов LINUX&UNIX необходимо иметь раздел на диске, CD-ROM или гибкий диск, который вы хотите подключить. Следует убедиться также, что каталог (точка подключения), к которому вы хотите подключить файловую систему, действительно существует. Подключение файловой системы этот каталог не создает, он должен существовать до попытки подключения, иначе подключение окончится неудачей.

Для подключения файловых систем вручную используют команду:

```
mount [ -t type ] [ -o options ] device mountpoint
```

где **device** - физическое устройство, которое необходимо подключить;

а **mountpoint** - точка подключения.

Использовать команду **mount** может только супер-пользователь (**root**).

Ниже представлены возможные опции (*Таблица 7.2*).

Таблица 7.2 Опции **mount**

-f	Имитирует подключение файловой системы. Выполняет все действия, кроме системного вызова для настоящего подключения
-v	Подробный отчет. **mount** предоставляет дополнительную информацию о своих действиях
-wr	Подключает файловую систему с доступом для чтения и записи
-ro	Подключает файловую систему с доступом только для чтения
-t type	Указывает тип подключаемой файловой системы. Допустимыми являются типы: ext2,msdos,hpfs,nfs,iso9660
-a	Указывает **mount** подключить все файловые системы, перечисленные в файле /etc/fstab
-o options list	Указывает mount применить список опций к подключаемой файловой системе

Например, следующая команда подключает CD-ROM SCSI только для чтения с форматом файлов ISO 9660 в каталог **/mnt/cdrom**:

```
mount -r -t iso9660 /dev/sr0 /mnt/cdrom
```

А следующая команда подключает все файловые системы Network File System(nfs), перечисленные в файле **/etc/fstab**:

```
mount -vat nfs
```

Рассмотрим монтирование гибкого диска. Сначала выполните команду **ls /mnt/floppy** и убедитесь в том, что этот каталог пуст.

Затем подключите гибкий диск в эту точку, используя команду **mount**. Учитывайте, что файловая система, которую пользователь хочет монтировать, определена в файле **/etc/fstab**, можно задать только точку монтирования, остальное будет извлечено командой **mount** из файла **/etc/fstab**. Снова выполните команду **ls /mnt/floppy** и убедитесь, что содержимое дискеты теперь появилось в этом каталоге.

Если правильно подключить файловую систему не удается, воспользуйтесь командой **mount -vf device mountpoint**. Эта команда выполнит все действия, кроме подключения, и выдаст подробный отчет.

Для подключения файловых систем при загрузке используют конфигурационный файл **`/etc/fstab`** (***file system table***). Список используемых файловых систем изменяется редко, поэтому удобно подключать их при загрузке. Файловые системы перечисляются по одной в строке. Поля в строках разделяются пробелами или табуляцией (*Таблица 7.3*).

Таблица 7.3 Поля файла fstab

Файловая система	Подключаемое блочное устройство или удаленная файловая система
Точка подключения	Куда подключить файловую систему. Чтобы сделать систему невидимой в дереве каталогов используйте слово none
Тип	Указывает тип подключаемой файловой системы (*)
Опции подключения	Разделенный запятыми список опций должен содержать, по крайней мере, тип подключения
Периодичность резервного копирования	Указывает, как часто следует выполнять резервное копирование с помощью команды dump. Если это поле отсутствует, то файловая система не нуждается в резервном копировании
Номер прохода	Задает порядок проверки целостности файловых систем при загрузке с помощью команды **fsck**. Для корневой файловой системы надо указывать 1, для остальных 2. Если значение не указано, то целостность файловой системы при загрузке проверяться не будет

(*) В настоящее время поддерживаются файловые системы следующих типов:

- **Minix** - локальная файловая система с именами файлов до 14 или 30 символов;
- **Ext2** - локальная файловая система с длинными именами файлов и другими возможностями;
- **Msdos** - локальная файловая система для разделов MS DOS;

- **Hpfs** - локальная файловая система для разделов OS/2 (***High Performance File System***);
- **Iso9660** - локальная файловая система, используемая с CD-ROM;
- **Nfs** - файловая система для подключения разделов удаленных систем;
- **Swap** - раздел или файл подкачки.

Приведем пример **fstab**:

# device	directory	type	options
/dev/hda1	/	ext2	defaults
/dev/hda2	/usr	ext2	defaults
/dev/hda2	none	swap	sw

Слово **`defaults`** указывает, что при подключении файловой системы следует применить набор опций по умолчанию:

- Файловую систему следует подключить с разрешенным доступом для чтения и записи;
- Она будет рассматриваться как отдельное блочное устройство;
- Весь файловый ввод/вывод будет выполняться асинхронно;
- Разрешено выполнение программных файлов;
- Файловая система может подключаться с помощью команды **`mount -a`**;
- Биты UID и GID интерпретируются в этой файловой системе;
- Обычным пользователям не разрешено подключать эту файловую систему.

Отключение файловых систем

Для отключения файловых систем используют команду:

`umount` `device или mountpoint`

где **`device`** - физическое устройство, которое необходимо отключить;

а **`mountpoint`** - точка подключения.

Отключает все файловые системы команда **umount -a**.

Команда **umount -t fstype** отключает только файловые системы указанного типа.

Рассмотрим размонтирование гибкого диска:

```
cd /mnt/floppy
pwd
ls -l
```

Попробуйте отключить гибкий диск, используя команду **umount**. Учитывайте, что, так как файловая система, которую вы хотите отключить, используется в этот момент, то появится сообщение об ошибке. Перейдите в каталог другой файловой системы и попробуйте снова.

Сетевая файловая система

Сетевая файловая система (***Network File System, NFS***) позволяет подключать файловые системы других компьютеров, используя протокол TCP/IP. Под NFS файловая система удаленного компьютера подключается и выглядит для пользователей как локальная файловая система. Для этого необходимо:

- Компьютеры должны быть способны связаться друг с другом по протоколу TCP/IP;
- Компьютер с файловой системой, которую хотим подключать по сети, должен предоставить ее для подключения. Такой компьютер называется сервером, а процесс предоставления - ***экспортом файловой системы***;
- Компьютер, подключающий экспортированную файловую систему, должен подключить ее как файловую систему типа NFS через файл **/etc/fstab** во время загрузки или вручную с помощью команды **mount**. Такой компьютер называется ***клиентом***.

Перед тем как файловая система будет предоставлена для сетевого подключения, она должна быть смонтирована на сервере, т.е. подключена локально.

Экспорт файловой системы

На сервере должны быть запущены демоны NFS **rpc.mountd** и **rpc.nfsd**. Их следует запускать, после того как был запущен **portmap**.

Запись об экспортируемой файловой системе должна быть внесена в конфигурационный файл **/etc/export**. Этот файл используется демонами **rpc.mountd** и **rpc.nfsd**, чтобы определить какие файловые системы экспортируются и какие для них существуют ограничения. Файловые системы перечисляются в этом файле по одной в строке.
В начале каждой строки указывается имя точки подключения локальной файловой системы, а затем список компьютеров, которым разрешается доступ к ней. После имени компьютера в круглых скобках через запятую может следовать список опций подключения. опции подключения, допустимые в /etc/export, представлены в таблице (*Таблица 7.4*).

Таблица 7.4 Опции подключения, допустимые в /etc/export

insecure	Разрешается доступ с этого компьютера без аутентификации
secure	Для доступа с этого компьютера требуется аутентификация
Root_squash	Все запросы от root с UID=0 на клиенте отображаются на UID nobody на сервере
no_root_squash	Запросы от UID 0 не отображаются
ro	Файловая система подключается только для чтения
rw	Файловая система подключается для чтения и записи
all_squash	Все UID и GID отображаются на анонимного пользователя

Приведем пример файла **export**:

/home white.tristar.com(rw) red.tristar.com(rw)

/projects brown.tristar.com(ro)

/public (ro,insecure,root_squash)

Список компьютеров, которым разрешен доступ к **/public** не указан. Это означает, что подключать ее может любой.

Подключение файловой системы NFS

Подключение можно выполнить через файл **/etc/fstab** во время начальной загрузки или вручную с помощью команды **mount**. Имя файловой системы должно быть указано в следующем формате:

```
Servername:filesystemname
```

где **Servername** - имя сервера, экспортирующего файловую систему,

а **Filesystemname** - имя файловой системы на сервере.

Опции подключения NFS представлены в таблице (*Таблица 7.5*).

Таблица 7.5 Опции подключения, применяемые NFS

rsize=n	Задает размер датаграмм (в байтах), используемых клиентом NFS в запросах на чтение. По умолчанию 10246айта
wsize=n	Задает размер датаграмм (в байтах), используемых клиентом NFS в запросах на запись. По умолчанию 10246айта
timeo=n	Устанавливает время ожидания выполнения запроса клиентом NFS. По умолчанию 0,7сек
hard	Выполняет жесткое подключение этой файловой системы
soft	Выполняет мягкое подключение этой файловой системы
intr	Разрешает прерывать вызов NFS

Пример **fstab**:

device directory type options

mailserver:/mail /mail nfs timeo=20, intr

Можно использовать команду **mount**:

```
mount -t nfs -o timeo=20, intr mailserver:/mail  /mail
```

Поддержка файловых систем

Файловые системы полезно проверять на наличие поврежденных или разрушенных файлов. Для этого используется команда **fsck** (***file system check***):

```
fsck [-A] [-V] [-t fstype] [-a] [-l] [-r] [-s] filesystem
```

Чаще всего используется команда **fsck filesystem.**.

В таблице (*Таблица 7.6*) представлены опции команды **fsck.**

Таблица 7.6 Опции fsck

-A	Проверяет все файловые системы, указанные в /etc/fstab
-V	Подробный отчет
-t fstype	Задает тип проверяемой файловой системы
Filesystem	Задает файловую систему. Можно указать как блочное устройство так и точку подключения
-a	Автоматически исправляет все ошибки
-l	Выводит список имен всех файлов
-r	Запрашивает подтверждение перед исправлением
-s	Выводит содержимое суперблока

Индекс

Л

М

Н

О

П

P

C

T

У

Ф

Ч

Э

Я

B

C

D

F

I

K

L

N

O

P

R

S

T

U

V

W