

Министерство образования и науки Российской Федерации

САНКТ-ПЕТЕРБУРГСКИЙ
ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО

Е.В. Душутина

**СИСТЕМНОЕ ПРОГРАММНОЕ
ОБЕСПЕЧЕНИЕ
МЕЖПРОЦЕССНЫЕ ВЗАИМОДЕЙСТВИЯ
В ОПЕРАЦИОННЫХ СИСТЕМАХ**

Учебное пособие

ИЗДАТЕЛЬСТВО
ПОЛИТЕХНИЧЕСКОГО УНИВЕРСИТЕТА

Санкт-Петербург
2016 г.

УДК 004.45 (УДК 681.3.06)
ББК 32.973.26-018.2
Д86

Р е ц е н з е н т – доктор технических наук,
профессор СПбПУ *В.Ф. Мелехин*

Душутина Е.В. **Системное программное обеспечение. Межпроцессные взаимодействия в операционных системах:** учеб. пособие. – СПб.: Изд-во Политехн. ун-та, 2016. – 180 с.

Учебное пособие предназначено для изучения дисциплин «Системное программное обеспечение» студентами, обучающимися по направлению 27.03.04 «Управление в технических системах» (ФГОС), а также для изучения дисциплин цикла «Операционные системы» и «Системное программирование» направления «Информатика и вычислительная техника» 09.03.01 и 09.04.01.

Излагаются основы управления процессами и потоками, их взаимодействия и синхронизации в операционных системах. Особое внимание уделяется практическим вопросам программирования средствами операционных систем, с использованием API и низкоуровневых средств программирования. В качестве базовых рассматриваются ОС семейств UNIX, Windows. Приводится большое количество практических примеров с их подробным разбором, анализом и результатами функционирования. Представленные программные фрагменты могут быть использованы для составления более сложных приложений, предлагаемых студентам для самостоятельной разработки в качестве заданий для закрепления полученных теоретических и практических навыков.

Учебное пособие также будет полезно специалистам, занимающимся разработкой системного программного обеспечения.

Печатается по решению
Совета по издательской деятельности Ученого совета
Санкт-Петербургского политехнического университета Петра Великого.

© Душутина Е.В., 2016
© Санкт-Петербургский политехнический
университет Петра Великого, 2016

ISBN 978-5-7422-5401-0

Оглавление

Введение	3
Системное программирование в ОС семейства Unix	
Раздел 1. Средства межпроцессного взаимодействия в ОС семейства Unix	4
1. Сигналы	
1.1. Ненадежные сигналы	8
1.2. Надежные сигналы	11
1.3. Сигналы реального времени	13
Каналы	
2. Неименованные каналы	15
3. Именованные каналы	18
4. Очереди сообщений	25
Количественные ограничения средств IPC	31
5. Семафоры и разделяемая память	32
6. Сокеты	44
Заключение по разделам ОС Unix	42
Системное программирование в ОС семейства Windows	
Раздел 2 . Управление процессами и потоками в Windows	51
2.1. Управление процессами	52
2.2. Управление потоками	62
2.3. Функции управления приоритетами процессов и потоков	67
Раздел 3. Средства межпроцессного взаимодействия в ОС Windows	81
3.1. Неименованные каналы	82
3.2. Именованные каналы	90
Сетевая передача данных с помощью именованных каналов	104
3.3. Использование сокетов	107
3.4. Сигналы в Windows	120
3.5. Разделяемая память	123
3. 6. Почтовые слоты	127
Раздел 4. Средства синхронизации потоков и процессов в ОС Windows и их применение	128
4.1. Использование мьютексов в качестве средства синхронизации	141
4.2. Семафоры для синхронизации в Windows	146
4.3. Критические секции	148
4.4. Объекты-события в качестве средства синхронизации	150
4.5. Условные переменные	153
4.6. Функции ожидания	156
4.7. Задача «читатели и писатели»	156
Заключение к разделам по Windows	177
Библиографический список	178

ВВЕДЕНИЕ

Процесс – базовое понятие ОС, часто кратко определяется как программа в стадии выполнения, при этом *программа* – это статический объект, представляющий собой файл с кодами и данными, тогда как *процесс* – это динамический объект, который возникает в ОС после «запуска» программы на выполнение и предоставления необходимых для этого ресурсов. Иногда процесс определяют как самодостаточную, но не минимальную, единицу вычислительной работы по преобразованию входных данных в выходные.

В большинстве многозадачных ОС процессы изолированы друг от друга, каждому из них предоставлены собственные ресурсы, прежде всего отдельное виртуальное адресное пространство, так, чтобы ни один из процессов не имел прямого доступа к адресному пространству другого процесса. Для взаимодействия друг с другом процессы обращаются к ОС, которая, выполняя функции посредника, предоставляет им специальные *средства межпроцессного взаимодействия* (IPC – InterProcess Communications).

В современных ОС процесс не является неделимой минимальной единицей работы. Существует понятие *потока* или *нити* (соответствуют одному англоязычному термину «thread») для определения вычислительных работ, входящих в состав процесса и разделяющих между собой ресурсы системы. Во многих ОС процессы являются лишь контейнерами для потоков, предоставляя им общие ресурсы, тогда как активными сущностями (выполняющими действия по обработке данных или преобразованию входной информации в выходную) являются именно потоки. Все потоки одного процесса используют общие файлы, таймеры, устройства, область ОЗУ, иные адресные пространства, могут иметь доступ к общему стеку. Кроме того, каждый из потоков/нитей имеет свой уникальный контекст.

Подсистема управления процессами планирует выполнение процессов, то есть распределяет процессорное время между несколькими одновременно существующими в системе процессами, а также занимается созданием и уничтожением процессов, обеспечивает процессы необходимыми системными ресурсами, поддерживает взаимодействие между процессами. Управление процессами (состав, контекст, инфраструктура, порождение, планирование и иные аспекты) подробно излагаются в (9). В данной работе основное внимание уделено взаимодействию процессов и потоков в многозадачных ОС на примере двух наиболее широко распространенных семейств ОС общего назначения UNIX и Windows. Вопросы синхронизации разбираются на примере средств Windows, т.к. отличаются бо́льшим разнообразием по сравнению с UNIX-системами.

Системное программирование в ОС семейства UNIX

Раздел 1. Средства межпроцессного взаимодействия в UNIX-подобных ОС

В Unix-подобных ОС средства межпроцессных взаимодействий (IPC – Inter Process Communication) включают следующий базовый набор:

- сигналы,
- анонимные (неименованные) каналы (иногда их называют программные),
- именованные каналы,
- очереди сообщений,
- семафоры
- разделяемая память,
- сокеты.

IPC в той или иной степени совмещают функции уведомления о событии и его обработки, передачи информации и синхронизации. В Unix-подобных ОС *все перечисленные средства, за исключением сокетов, являются локальными*, т.к. используют буферизацию в пространстве ядра и адресуются в локальном пространстве памяти.

Для исследования IPC на примерах используем ОС Linux (Ubuntu 14.04).

1. Сигналы

Сигналы позволяют осуществить самый примитивный способ коммуникации между двумя процессами. Сигналы в системе UNIX используются для того, чтобы: сообщить процессу о том, что возникло асинхронное событие; или необходимо обработать исключительное состояние.

Изначально сигналы были разработаны для уведомления об ошибках. В дальнейшем их стали использовать и как простейшую форму межпроцессного взаимодействия (IPC), например, для синхронизации процессов или для передачи простейших команд от одного процесса другому.

Сигнал позволяет передать уведомление о некотором произошедшем событии между процессами или между ядром системы и процессами. Это означает, что посредством сигналов можно выполнять две основные функции IPC: передачу информации и синхронизацию процессов или потоков.

Для отправки и доставки сигнала требуется системный вызов. Для доставки – прерывание и его обработка. При этом требуется проведение довольно большого числа операций со стеком – копирование пользовательского стека в системную область, извлечение параметров и результатов работы системных вызовов и прерываний. Поскольку объем передаваемой информации при этом способе взаимодействия не велик, а затраты на его реализацию существенны, сигналы считаются одним из самых ресурсоемких способов IPC.

Каждый сигнал имеет уникальное символьное имя и соответствующий ему номер. Базовый перечень сигналов, поддерживаемый практически в любой POSIX-ориентированной ОС, составляет не более тридцати двух (количество бит в тридцати двух-разрядном слове) и в большинстве современных систем их номера смещены к началу нумерации. Наряду с базовыми в POSIX ОС дополнительно может поддерживаться свой уникальный набор сигналов.

Кроме того, с расширением стандарта POSIX и современными возможностями наращивания разрядной сетки (до шестидесяти четырех) перечень сигналов во многих ОС тоже расширился. Появился еще один тип сигналов – сигналы реального времени, которые могут принимать значения между SIGRTMIN и SIGRTMAX включительно. POSIX требует, чтобы предоставлялось по крайней мере RTSIG_MAX сигналов, и минимальное значение этой константы равно 8.

Ознакомиться с полным перечнем сигналов можно с помощью команды *kill -l* в командном интерпретаторе той реализации ОС, с которой вы работаете, например, один из возможных вариантов:

```
$ kill -l
1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL 5) SIGTRAP
6) SIGABRT 7) SIGBUS 8) SIGFPE 9) SIGKILL 10) SIGUSR1
11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP
21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO 30) SIGPWR
31) SIGSYS 34) SIGRTMIN 35) SIGRTMIN+1 36) SIGRTMIN+2
37) SIGRTMIN+3 38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6
41) SIGRTMIN+7 42) SIGRTMIN+8 43) SIGRTMIN+9 44) SIGRTMIN+10
45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13 48) SIGRTMIN+14
49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8
57) SIGRTMAX-7 58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4
61) SIGRTMAX-3 62) SIGRTMAX-2 63) SIGRTMAX-1 64) SIGRTMAX
```

Следует заметить, что именование базовых сигналов, как правило, совпадает в разных Unix-подобных ОС, чего нельзя сказать о нумерации, поэтому целесообразно сначала ознакомиться со списком.

Кроме того, сигнал может быть отправлен процессу либо ядром, либо другим процессом с помощью системного вызова `kill()`:

```
#include <signal.h>
int kill(pid_t pid, int sig);
```

Аргумент *pid* адресует процесс, которому посылается сигнал. Аргумент *sig* определяет тип отправляемого сигнала. С помощью системного вызова *kill()* процесс может послать сигнал, как самому себе, так и другому процессу или группе процессов. В этом случае процесс, посылающий сигнал, должен иметь те же реальный и эффективный идентификаторы, что и процесс, которому сигнал отправляется. Разумеется, данное ограничение не распространяется на ядро или процессы, обладающие привилегиями суперпользователя. Они имеют возможность отправлять сигналы любым процессам системы.

Аналогичное действие можно произвести из командной строки в терминальном режиме, используя команду интерпретатора

```
kill pid
```

К генерации сигнала могут привести различные ситуации:

1. Ядро отправляет процессу (или группе процессов) сигнал при нажатии пользователем определенных клавиш или их комбинаций.

2. Аппаратные особые ситуации, например, деление на 0, обращение недопустимой области памяти и т.д., также вызывают генерацию сигнала. Обычно эти ситуации определяются аппаратурой компьютера, и ядру посылается соответствующее уведомление (например, виде прерывания). Ядро реагирует на это отправкой соответствующего сигнала процессу, который находился в стадии выполнения, когда произошла особая ситуация.

3. Определенные программные состояния системы или ее компонентов также могут вызвать отправку сигнала. В отличие от предыдущего случая, эти условия не связаны с аппаратной частью, а имеют программный характер. В качестве примера можно привести сигнал SIGALRM, отправляемый процессу, когда срабатывает таймер, ранее установленный с помощью вызова `alarm()`.

Процесс может выбрать одно из трех возможных действий при получении сигнала:

1. игнорировать сигнал,
2. перехватить и самостоятельно обработать сигнал,
3. позволить действие по умолчанию.

Текущее действие при получении сигнала называется **диспозицией сигнала**.

Порожденный вызовом *fork()* процесс наследует диспозицию сигналов от своего родителя. Однако при вызове *exec()* диспозиция всех перехватываемых сигналов будет установлена ядром на действие *по умолчанию*.

В ОС поддерживается ряд функций, позволяющих управлять диспозицией сигналов.

Наиболее простой в использовании является функция *signal()*. Она позволяет устанавливать и изменять диспозицию сигнала.

```
#include <signal.h>
void (*signal (int sig, void (*disp)(int)))(int);
```

Аргумент *sig* определяет сигнал, диспозицию которого нужно изменить. Аргумент *disp* определяет новую диспозицию сигнала. Возможны следующие три варианта:

Значение	Назначение
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. Не все сигналы можно игнорировать.
Имя ф-ции	Указывает на определенную пользователем функцию-обработчик

Возвращаемое функцией *signal()* значение может быть различным в различных ОС. В UNIX-подобных ОС, как правило, в случае успешного завершения *signal()* возвращает предыдущую диспозицию - это может быть функция-обработчик сигнала или системные значения SIG_DFL или SIG_IGN. Это значение в случае необходимости может быть использовано для восстановления предыдущей диспозиции после однократного выполнения пользовательского обработчика. Для многократного использования требуется предусмотреть повторный вызов *signal()* в теле обработчика.

Более гибкое управление сигналами предоставляет функция *sigaction()*:

```
int sigaction( int sig,
               const struct sigaction * act,
               struct sigaction * oact );
```

Данная функция позволяет вызывающему процессу получить информацию или установить (или и то и другое) действие, соответствующее какому-либо сигналу или группе сигналов. При этом каждый сигнал ассоциируется с битом 32-х/(64-х) –разрядного слова-маски, соответствующим номеру сигнала.

Аргумент *sig* определяет тип сигнала (все типы сигналов определены в библиотеке *signal.h*).

Аргумент *act* – если он не нулевой, то действие при указанном сигнале изменятся.

Аргумент *oact* - если он не нулевой, то предыдущее действие при указанном сигнале сохраняется в структуре типа *sigaction*, на которую указывает указатель *oact*.

Комбинация *act* и *oact* позволяет запрашивать или устанавливать новые действия при поступлении сигнала.

Состав структуры *sigaction*:

<code>void (*sa_handler)()</code>	адрес обработчика сигнала или действие для незапрашиваемых сигналов
<code>void (*sa_sigaction)(int signo, siginfo_t* info, void* other)</code>	адрес обработчика сигнала или действие для запрашиваемых сигналов
<code>sigset_t sa_mask</code>	дополнительный набор сигналов, который будет заблокирован при обработке поступившего сигнала
<code>int sa_flags</code>	специальные флаги для воздействия на режим работы сигнала

Компоненты *sa_handler* и *sa_sigaction* структуры *sigaction* вызываются со следующими аргументами:

```
void handler(int signo, siginfo_t* info, void* other);
```

если обработчик сигнала представлен: `void handler(int signo),`

то аргументы `siginfo_t* info` и `void* other` будут игнорироваться.

Для работы с сигналами реального времени существует дополнительный набор функций.

Выбор функции управления сигналами определяет свойства сигнала как средства IPC: *signal()* обеспечивает так называемую *ненадежную* передачу сигнала, тогда как *sigaction()* гарантирует *надежную* передачу. Последнее означает, что если при возникновении сигнала система занята обработкой другого сигнала (назовем его «текущим»), то возникший сигнал не будет потерян, а его обработка будет отложена до окончания текущего обработчика.

Разберем эти свойства более подробно **на примерах**.

1.1. Ненадежные сигналы

Пример. ТЗ: Создать программу, позволяющую изменить диспозицию сигналов, а именно, установить:

- обработчик пользовательских сигналов SIGUSR1 и SIGUSR2;
- реакцию по умолчанию на сигнал SIGINT;
- игнорирование сигнала SIGCHLD;

Породить процесс-копию и уйти в ожидание сигналов. Обработчик сигналов должен содержать восстановление диспозиции и оповещение на экране о

полученном (удачно или неудачно) сигнале и идентификаторе родительского процесса. Процесс-потомок, получив идентификатор родительского процесса, должен отправить процессу-отцу сигнал SIGUSR1 и извещение об удачной или неудачной отправке указанного сигнала. Остальные сигналы можно сгенерировать из командной строки.

Исходный код (в файле sigExam.c):

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

static void sigHandler(int sig) {
    printf("Caught signal %s\n", sig == SIGUSR1 ? "SIGUSR1": "SIGUSR2");
    printf("Parent = %d\n", getppid());
    // восстанавливаем старую диспозицию
    signal(sig, SIG_DFL);
}

int main() {
    printf("\nFather started: pid = %i, ppid = %i\n", getpid(), getppid());
    signal(SIGUSR1, sigHandler);
    signal(SIGUSR2, sigHandler);
    signal(SIGINT, SIG_DFL);
    signal(SIGCHLD, SIG_IGN);
    int forkRes = fork();
    if(forkRes == 0) {
        // программа-потомок
        printf("\nSon started: pid = %i, ppid = %i\n", getpid(), getppid());
        // отправляем сигналы родителю
        if(kill(getppid(), SIGUSR1) != 0) {
            printf("Error while sending SIGUSR1\n");
            exit(1);
        }
        printf("Successfully sent SIGUSR1\n");
        return 0;
    }
    // программа-родитель
    wait(NULL);
    // ждем сигналов
    for(;;){
        pause();
    }
    return 0;
}
```

Обе программы (родителя и потомка) зададим в одном файле. С одной стороны, это делает код более компактным, с другой – упрощает наследование за счет использования только *fork()* –вызова и позволяет потомку скопировать диспозицию родителя.

Ветвление происходит сразу же за вызовом *fork()*. Если он вернул 0, значит,

выполняется код программы-сына, иначе — код программы отца.

Скомпилируем и выполним программу:

```
de@de:~/lab4$ cc -o sigExam sigExam.c
de@de:~/lab4$ ./sigExam

Father started: pid = 14589,ppid = 12231

Son started: pid = 14590,ppid = 14589
Successfully sent SIGUSR1
Catched signal SIGUSR1
Parent = 12231
```

Процесс-потомок отправил сигнал SIGUSR1, а процесс-отец его успешно принял. Отправим еще 3 сигнала процессу-отцу: SIGCHLD, SIGUSR2, SIGINT:

```
de@de:~/lab4/sig$ kill -SIGUSR2 14589
de@de:~/lab4/sig$ kill -SIGCHLD 14589
de@de:~/lab4/sig$ kill -SIGINT 14589
Результат:
Catched signal SIGUSR2
Parent = 12231

de@de:~/lab4/sig$
```

Сигнал SIGUSR2 также был «пойман», на сигнал SIGCHLD не последовало никакой реакции (так как он был проигнорирован), и сигнал SIGINT привел к завершению работы.

Запустим программу еще раз и дважды отправим ей сигнал SIGUSR2:

```
de@de:~/lab4/sig$ ./sigExam

Father started: pid = 16225,ppid = 12231

Son started: pid = 16226,ppid = 16225
Successfully sent SIGUSR1
Catched signal SIGUSR1
Parent = 12231
Catched signal SIGUSR2
Parent = 12231
User defined signal 2
```

В результате первый сигнал был «пойман», второй обработался по умолчанию. Это происходит потому, что в обработчике прерываний после первого приема сигнала происходит восстановление диспозиции сигналов. Аналогичная ситуация была бы при двукратной отправке процессу сигнала SIGUSR1.

Самостоятельно

повторите эксперимент для других сигналов и процессов, порождаемых в

разных файлах; а также для потоков одного и разных процессов.

1.2. Надежные сигналы

Пример.

ТЗ: Создать программу, позволяющую продемонстрировать возможность отложенной обработки (временного блокирования) сигнала (например, SIGINT).

Вся необходимая для управления сигналами информация передается через указатель на структуру *sigaction*. Блокировку реализуем, вызвав "засыпание" процесса на одну минуту из обработчика пользовательских сигналов. В основной программе установим диспозицию этих сигналов. С рабочего терминала отправим процессу *sigact* сигнал SIGUSR1 или SIGUSR2, а затем сигнал SIGINT.

Исходный код (sigact.c):

```
#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>

void (*mysig(int sig, void (*hnd)(int)))(int) {
    // надежная обработка сигналов
    struct sigaction act, oldact;
    act.sa_handler = hnd;
    sigemptyset(&act.sa_mask);
    sigaddset(&act.sa_mask, SIGINT);
    act.sa_flags = 0;
    if(sigaction(sig, &act, 0) < 0)
        return SIG_ERR;
    return act.sa_handler;
}

void hndUSR1(int sig) {
    if(sig != SIGUSR1) {
        printf("Caught bad signal %d\n", sig);
        return;
    }
    printf("SIGUSR1 caught\n");
    sleep(60);
}

int main() {
    mysig(SIGUSR1, hndUSR1);
    for(;;) {
        pause();
    }
    return 0;
}
```

```

}
Результаты выполнения:
de@de:~/lab4/sig$ cc -w -o sigact sigact.c
de@de:~/lab4/sig$ ./sigact &
[1] 25329
de@de:~/lab4/sig$ kill -SIGUSR1 %1
SIGUSR1 caught
de@de:~/lab4/sig$ kill -SIGINT %1
de@de:~/lab4/sig$ jobs
[1]+  Running                  ./sigact &
de@de:~/lab4/sig$ jobs
[1]+  Interrupt                ./sigact

```

Чтобы иметь возможность отправить сигналы с терминала следует запустить программу в фоновом режиме. По результатам сигнал SIGUSR1 принят корректно, но после посылки сигнала SIGINT программа продолжала выполняться еще минуту, и только после этого завершилась. В этом отличие надежной обработки сигналов от ненадежной: есть возможность отложить прием некоторых других сигналов. Отложенные таким образом сигналы записываются в маску PENDING и обрабатываются после завершения обработки сигналов, которые отложили обработку. Механизм ненадежных сигналов не позволяет откладывать обработку других сигналов (можно лишь установить игнорирование некоторых сигналов на время обработки).

Пример

ТЗ: Изменить обработчик сигнала так, чтобы из него производилась отправка другого сигнала.

Пусть из обработчика сигнала SIGUSR1 функцией kill() генерируется сигнал SIGINT. Проанализируем наличие и очередность обработки сигналов.

Исходный код программы:

```

#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>

void (*mysig(int sig,void (*hnd)(int)))(int) {
    // надежная обработка сигналов
    struct sigaction act,oldact;
    act.sa_handler = hnd;
    sigemptyset(&act.sa_mask);
    sigaddset(&act.sa_mask,SIGINT);

```

```

    act.sa_flags = 0;
    if(sigaction(sig,&act,0) < 0)
        return SIG_ERR;
    return act.sa_handler;
}
void hndUSR1(int sig) {
    if(sig != SIGUSR1) {
        printf("Caught bad signal %d\n",sig);
        return;
    }
    printf("SIGUSR1 caught, sending SIGINT\n");
    kill(getpid(),SIGINT);
    sleep(10);
}
int main() {
    mysig(SIGUSR1,hndUSR1);
    for(;;) {
        pause();
    }
    return 0;
}

```

Результаты выполнения программы:

```

de@de:~/lab4/sig$ cc -w -o sigact2 sigact2.c
de@de:~/lab4/sig$ ./sigact2 &
[1] 28822
de@de:~/lab4/sig$ kill -SIGUSR1 %1
de@de:~/lab4/sig$ SIGUSR1 caught, sending SIGINT
jobs
[1]+  Running                  ./sigact2 &
de@de:~/lab4/sig$ jobs
[1]+  Running                  ./sigact2 &
de@de:~/lab4/sig$ kill -SIGINT %1
[1]+  Interrupt                ./sigact2

```

При генерации сигнала (в данном случае SIGINT) из обработчика другого сигнала обработка сгенерированного сигнала задерживается до конца выполнения текущего обработчика (как и в предыдущем эксперименте).

1.3. Сигналы POSIX реального времени

Некоторые реализации POSIX ОС могут обрабатывать все сигналы как сигналы реального времени, но для UNIX-подобных ОС это не является обязательным. Если мы хотим, чтобы сигналы *гарантированно* обрабатывались как сигналы реального времени, мы должны:

- использовать сигналы с номерами в диапазоне от SIGRTMIN до SIGRTMAX
- должны указать флаг SA_SIGINFO при вызове *sigaction()* с установкой обработчика сигнала

- обработчик сигнала реального времени, устанавливаемый с флагом SA_SIGINFO, объявляется как:

```
void func(int signo, siginfo_t *info, void *context); где
signo— номер сигнала,
siginfo_t — структура, определяемая как
typedef struct {
    int si_signo; /* то же, что и signo */
    int si_code; /* SI_{USER,QUEUE,TIMER,ASYNCIO,MESGQ} */
    union sigval si_value; /* целое или указатель от отправителя */
} siginfo_t;
```

на что указывает *context* — зависит от реализации.

Таким образом, сигналы реального времени несут больше информации, чем прочие сигналы (при отправке сигнала, не обрабатываемого как сигнал реального времени, единственным аргументом обработчика является номер сигнала).

- SIGRTMIN и SIGRTMAX – это еще и макросы (вызывающие *sysconf*), которые позволяют изменять сами эти значения.

«Характеристики сигналов реального времени» означает следующее:

- Сигналы помещаются в очередь.
- Если сигнал будет порожден несколько раз, он будет несколько раз получен адресатом. Более того, повторения одного и того же сигнала доставляются в порядке очереди (FIFO). Если же сигналы в очередь не помещаются, неоднократно порожденный сигнал будет получен лишь один раз.
- Когда в очередь помещается множество неблокируемых сигналов в диапазоне SIGRTMIN—SIGRTMAX, сигналы с меньшими номерами доставляются раньше сигналов с большими номерами. То есть сигнал с номером SIGRTMIN имеет «большой приоритет», чем сигнал с номером SIGRTMIN+1, и т.д.

Самостоятельно

- 1) проведите эксперимент, позволяющий определить возможность организации очереди для различных типов сигналов, обычных и реального времени, (более двух сигналов, для этого увеличьте «вложенность» вызовов обработчиков);
- 2) экспериментально подтвердите, что обработка равноприоритетных сигналов реального времени происходит в порядке FIFO;

- 3) опытным путем подтвердите наличие приоритетов сигналов реального времени.

Каналы

Различают два типа каналов анонимные (иначе их называют «программные» или «неименованные») и именованные. Они по-разному реализованы, но доступ к ним организуется одинаково с помощью обычных функций `read` и `write` (унифицированный подход по типу *файловой модели*). Одним из свойств программных каналов и FIFO является то, что данные по ним передаются в виде потоков байтов (аналогично соединению TCP). Деление этого потока на самостоятельные записи полностью предоставляется приложению (в отличие, например, от очередей сообщений, которые автоматически расставляют границы между записями, аналогично тому, как это делается в дейтаграммах UDP).

Рассмотрим оба типа каналов более детально с примерами программ.

2. Неименованные каналы

Программные (неименованные) каналы – однонаправленные, используются только для связи родственных процессов, в принципе могут использоваться и неродственными процессами, если предоставить им возможность передавать друг другу дескрипторы (т.к. имен они не имеют). Неименованный канал создается посредством системного вызова `pipe(2)`, который возвращает 2 файловых дескриптора `filedes[1]` для записи в канал и `filedes[0]` для чтения из канала:

```
#include
int pipe(int fd[2]);
/* возвращает 0 в случае успешного завершения. -1 – в случае ошибки:*/
```

Доступ к дескрипторам канала может получить как процесс, вызвавший `pipe()`, так и его дочерние процессы. Канал создается одним процессом, может использоваться им единолично (но редко). Как правило, это средство применяется для связи между двумя процессами, следующим образом: процесс создает канал, а затем вызывает `fork()`, создавая свою копию — дочерний процесс. Затем родительский процесс закрывает открытый для чтения конец канала, а дочерний, в свою очередь, — открытый на запись конец канала. Это обеспечивает одностороннюю передачу данных между процессами. При необходимости передачи данных в обе стороны нужно создавать пару каналов и использовать каждый из них для передачи данных в одну сторону.

Пример.

ТЗ: Организуем программу (файл pipe.c) так, чтобы процесс-родитель создавал неименованный канал, создавал потомка, закрывал канал на запись и записывал в произвольный текстовый файл считываемую из канала информацию. В функции процесса-потомка будет входить считывание данных из файла и запись их в канал. (Функционирование осуществляется через стандартные потоки ввода/вывода, как было показано выше).

Алгоритм работы программы:

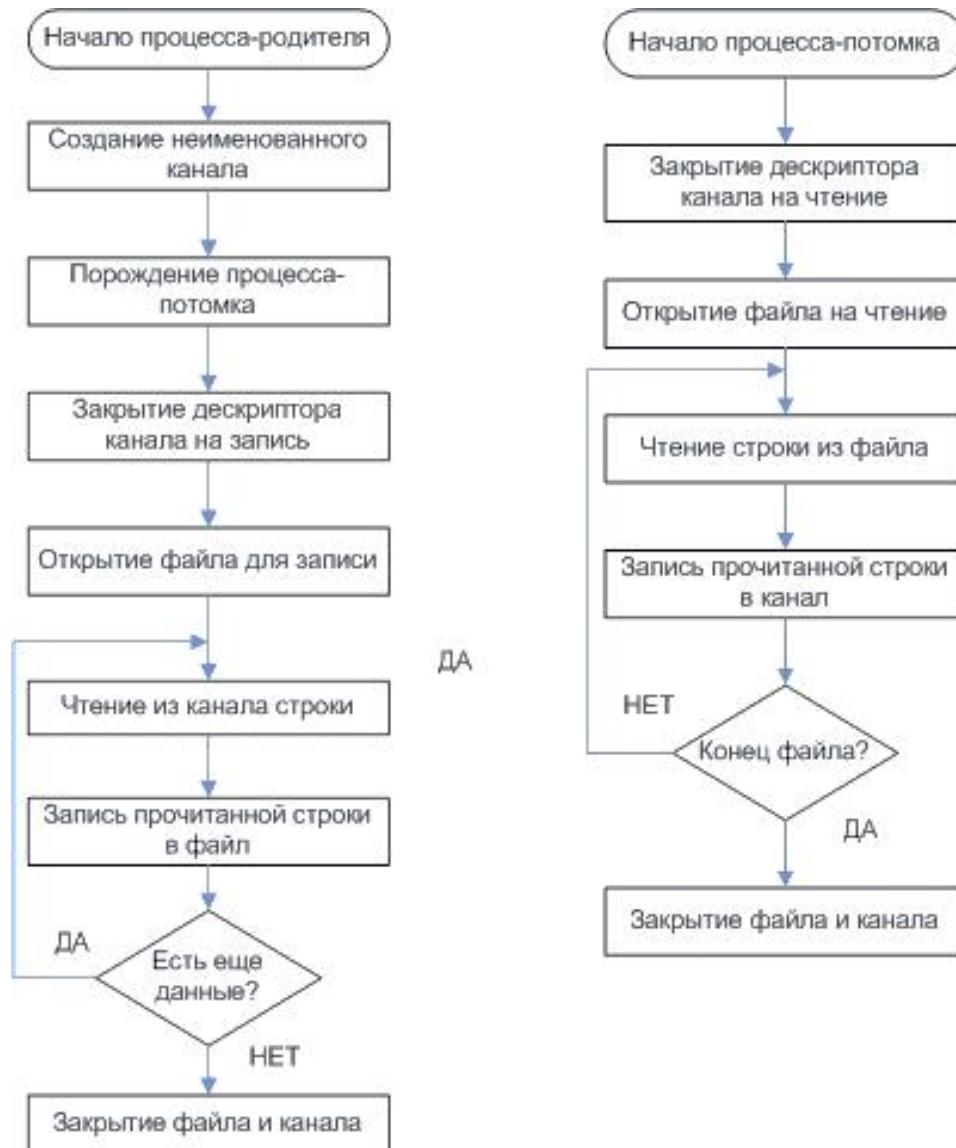


Рис. 1. Взаимодействие посредством неименованных каналов

Исходный текст программы (pipe.c):

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define DEF_F_R "from.txt"
#define DEF_F_W "to.txt"
int main(int argc, char** argv) {
    char fileToRead[32];
    
```

```

char fileToWrite[32];
if(argc < 3 ){
printf("Using default fileNames '%s', '%s'\n",DEF_F_R,DEF_F_W);
strcpy(fileToRead,DEF_F_R);
strcpy(fileToWrite,DEF_F_W);
}
else {
strcpy(fileToRead,argv[1]);
strcpy(fileToWrite,argv[2]);
}
int filedes[2];
if(pipe(filedes) < 0) {
printf("Father: can't create pipe\n");
exit(1);
}
printf("pipe is successfully created\n");
if(fork() == 0) {
// процесс сын
// закрывает пайп для чтения
close(filedes[0]);
FILE* f = fopen(fileToRead,"r");
if (!f) {
printf("Son: cant open file %s\n",fileToRead);
exit(1);
}
char buf[100];
int res;
while(!feof(f)) {
// читаем данные из файла
res = fread(buf,sizeof(char),100,f);
write(filedes[1],buf,res); // пишем их в пайп
}
close(f);
close(filedes[1]);
return 0;
}
// процесс отец
// закрывает пайп для записи
close(filedes[1]);
FILE *f = fopen(fileToWrite,"w");
if (!f) {
printf("Father: cant open file %s\n",fileToWrite);
exit(1);
}
char buf[100];
int res;
while(1) {
bzero(buf,100);
res = read(filedes[0],buf,100);
if(!res)
break;
printf("Read from pipe: %s\n",buf);
fwrite(buf,sizeof(char),res,f);
}

```

```
}  
fclose(f);  
close(filedes[0]);  
return 0;  
}
```

Результат выполнения:

```
dm@dm:~/lab4/pipe$ cc -w -o pipe pipe.c  
dm@dm:~/lab4/pipe$ ./pipe  
Using default fileNames 'from.txt','to.txt'  
pipe is successfully created  
Read from pipe: first string  
second  
third  
That's all  
dm@dm:~/lab4/pipe$
```

Содержимое файла from.txt:

```
first string  
second  
third  
That's all
```

Содержимое файла from.txt успешно переписалось в изначально пустой файл to.txt с использованием неименованного канала.

Содержимое файла to.txt после запуска программы:

```
first string  
second  
third  
That's all
```

Так как процесс-родитель только читает из канала, то дескриптор для записи (filedes[1]) он закрывает, аналогично процесс-сын в начале работы закрывает дескриптор для чтения из канала (filedes[0]).

Главное применение неименованных каналов в ОС Unix – реализация конвейеров команд в интерпретаторах командной строки.

Например, при выполнении конвейера

```
ls|sort|grep
```

интерпретатор создаст три процесса с двумя каналами между ними так, что открытый для чтения конец каждого канала будет подключен к стандартному потоку ввода, а открытый на запись — к стандартному потоку вывода.

3. Именованные каналы

Именованные каналы в Unix функционируют подобно неименованным — они позволяют передавать данные только в одну сторону. Однако в отличие от неименованных каналов каждому каналу FIFO сопоставляется полное имя в

файловой системе, что позволяет двум неродственным процессам обратиться к одному и тому же FIFO. Аббревиатура FIFO расшифровывается как «first in, first out» — «первым вошел, первым вышел», то есть эти каналы работают как очереди.

После создания канал FIFO должен быть открыт на чтение или запись с помощью либо функции `open`, либо одной из стандартных функций открытия файлов из библиотеки ввода-вывода (например, `fopen`). FIFO может быть открыт либо только на чтение, либо только на запись. Нельзя открывать канал на чтение и запись одновременно, поскольку именованные каналы могут быть только односторонними.

Пример.

ТЗ: Создать клиент-серверное приложение, демонстрирующее дуплексную (двунаправленную) передачу информации двумя однонаправленными именованными каналами между клиентом и сервером.

В файле `server.c` в основной программе:

создадим 2 именованных канала, используя системный вызов `mknod()`, аргументы которого: имя файла FIFO в файловой системе; флаги владения, прав доступа (установим открытые для всех права доступа на чтение и на запись `S_IFIFO | 0666`). Откроем один канал на запись (`chan1`), другой - на чтение (`chan2`) и запустим серверную часть программы.

В серверной части программы:

запишем имя файла в канал 1 (для записи) функцией `write()`; прочитаем данные из канала 2 и выведем на экран.

В файле `client.c` запрограммируем функции: открытия каналов для чтения (`chan1`) и записи (`chan2`). Из первого канал читается имя файла, во второй канал пишется его содержимое. Алгоритм работы показан ниже на рис. 2.

Исходный код программы `server.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define DEF_FILENAME "testFile.txt"
int main(int argc, char** argv) {
    char fileName[30];
    if(argc < 2) {
        printf("Using default file name '%s'\n", DEF_FILENAME);
        strcpy(fileName, DEF_FILENAME);
```

```

}
else
    strcpy(fileName,argv[1]);
// создаем два канала
int res = mknod("channel1",S_IFIFO | 0666,0);
if(res) {
    printf("Can't create first channel\n");
    exit(1);
}
res = mknod("channel2",S_IFIFO | 0666,0);
if(res) {
    printf("Can't create second channel\n");
    exit(1);
}
// открываем первый канал для записи
int chan1 = open("channel1",O_WRONLY);
if(chan1 == -1) {
    printf("Can't open channel for writing\n");
    exit(0);
}
// открываем второй канал для чтения
int chan2 = open("channel2",O_RDONLY);
if(chan2 == -1) {
    printf("Can't open channe2 for reading\n");
    exit(0);
}
// пишем имя файла в первый канал
write(chan1,fileName,strlen(fileName));
// читаем содержимое файла из второго канала
char buf [100];
for(;;) {
    bzero(buf,100);
    res = read(chan2,buf,100);
    if(res <= 0)
        break;
    printf("Part of file: %s\n");
}
close(chan1);
close(chan2);
unlink("channel1");
unlink("channel2");
return 0;
}

```

Исходный код программы client.c

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

```

```

int main() {
    // каналы сервер уже создал, открываем их
    int chan1 = open("channel1", O_RDONLY);
    if(chan1 == -1) {
        printf("Can't open channel1 for reading\n");
        exit(0);
    }
    int chan2 = open("channel2", O_WRONLY);
    if(chan2 == -1) {
        printf("Can't open channel2 for reading\n");
        exit(0);
    }
    // читаем имя файла из первого канала
    char fileName[100];
    bzero(fileName,100);
    int res = read(chan1,fileName,100);
    if(res <= 0) {
        printf("Can't read fileName from channel1\n");
        exit(0);
    }
    // открываем файл на чтение
    FILE *f = fopen(fileName,"r");
    if(!f) {
        printf("Can't open file %s\n",fileName);
        exit(0);
    }
    // читаем из файла и пишем во второй канал
    char buf[100];
    while(!feof(f)) {
        // читаем данные из файла
        res = fread(buf,sizeof(char),100,f);
        // пишем их в канал
        write(chan2,buf,res);
    }
    fclose(f);
    close(chan1);
    close(chan2);
    return 0;
}

```

Запустим программу *server* из одного терминала и *client* из другого:

Результат работы сервера:

```

de@de:~/lab4/fifo$ ./server
Using default file name 'testFile.txt'
Part of file: first string
second
Last string
de@de:~/lab4/fifo$

```

Результат работы клиента:

```
de@de:~/lab4/fifo$ ./client
```

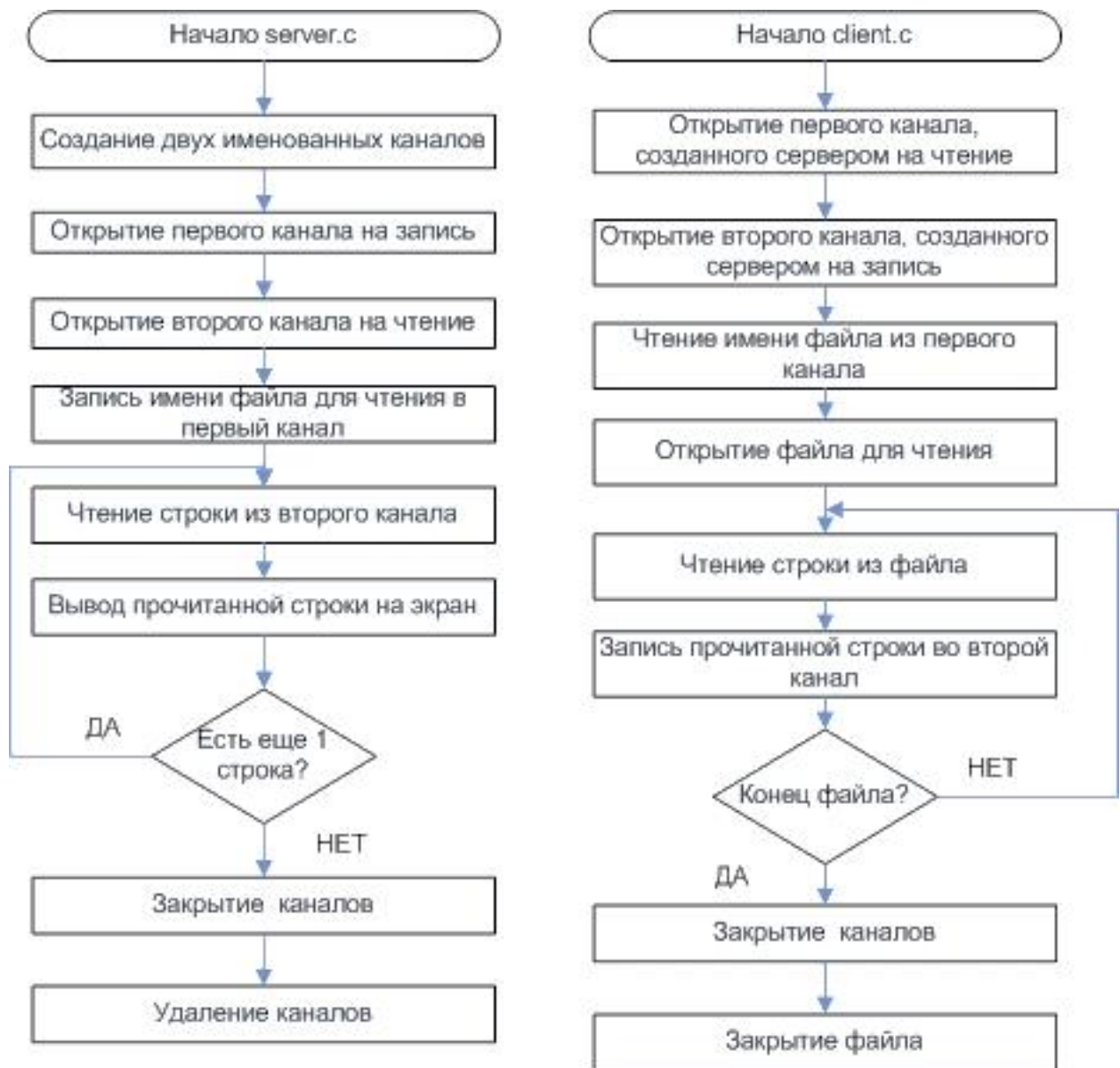


Рис.2. Взаимодействие посредством именованных каналов

В результате выполнения программы содержимое файла testFile.txt:

```
first string
second
Last string
```

Сервер создает два канала, записывает в один из них имя файла и ждёт данные от клиента. Каналы создаются в рабочей папке сервера, и использовать их может любой процесс, а не только дочерний по отношению к серверу. Клиент после запуска также открывает уже созданные каналы, считывает имя

файла и отправляет серверу его содержимое, используя второй канал. После завершения передачи, сервер уничтожает каналы с помощью функции `unlink()`.

Несмотря на то, что именованные каналы являются отдельным типом файлов и могут быть видимы разными процессами даже в распределенной файловой системе, использование FIFO для взаимодействия удаленных процессов и обмена информацией между ними невозможно. Так как и в этом случае для передачи данных задействовано ядро. Создаваемый файл служит для получения данных о расположении FIFO в адресном пространстве ядра и его состоянии.

Продемонстрируем это **на примере**. Изменим ранее использованную программу так, чтобы сервер, перед тем как читать данные из канала, ожидал ввода пользователя. Исходный код клиента оставим неизменным.

Исходный код сервера:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define DEF_FILENAME "testFile.txt"
int main(int argc, char** argv) {
    char fileName[30];
    if(argc < 2) {
        printf("Using default file name '%s'\n",DEF_FILENAME);
        strcpy(fileName,DEF_FILENAME);
    }
    else
        strcpy(fileName,argv[1]);
    // создаем два канала
    int res = mknod("channel1",S_IFIFO | 0666,0);
    if(res) {
        printf("Can't create first channel\n");
        exit(1);
    }
    res = mknod("channel2",S_IFIFO | 0666,0);
    if(res) {
        printf("Can't create second channel\n");
        exit(1);
    }
    // открываем первый канал для записи
    int chan1 = open("channel1",O_WRONLY);
    if(chan1 == -1) {
        printf("Can't open channel for writing\n");
        exit(0);
    }
}
```



```

}
// открываем второй канал для чтения
int chan2 = open("channel2",O_RDONLY);
if(chan2 == -1) {
    printf("Can't open channe2 for reading\n");
    exit(0);
}
// пишем имя файла в первый канал
write(chan1,fileName,strlen(fileName));
// читаем содержимое файла из второго канала
char buf [100];
printf("Waiting for clint write to channel\n");
getchar();
for(;;) {
    bzero(buf,100);
    res = read(chan2,buf,100);
    if(res <= 0)
        break;
    printf("Part of file: %s\n");
}
close(chan1);
close(chan2);
unlink("channel1");
unlink("channel2");
printf("Servr finished\n");
return 0;
}

```

Запустим клиент и сервер. Пока сервер ожидает ввода, посмотрим размер файла канала.

Сервер:

```

de@de:/lab4/4-fifodop$ ./server
Using default file name 'testFile.txt'
Waiting for clint write to channel

```

Клиент:

```

de@de:/lab4/4-fifodop$ ./client
Client finished
de@de:/lab4/4-fifodop$

```

Размер файла каналов:

```

de@de:/lab4/4-fifodop$ ls -sl | grep chan
0 prwxrwx--- 1 root plugdev 0 Dec 26 22:18 channel1
0 prwxrwx--- 1 root plugdev 0 Dec 26 22:18 channel2

```

Размер файла канала не изменяется, несмотря на записанные данные, это свидетельствует о том, что файл используется не как хранилище пересылаемых данных, а только для получения информации системой о них. Сами данные проходят через ядро ОС.

Позволим серверу выполняться дальше, нажав на *enter*, и убедимся, что данные получены:

```
Waiting for clint write to channel
```

```
Part of file: first string  
second  
Last string
```

```
Servr finished  
de@de:/lab4/4-fifodop$
```

В первую строку вывода умышленно внесены орфографические ошибки для идентификации работы измененного сервера.

Ограничения каналов

На неименованные каналы и каналы FIFO системой накладываются всего **два ограничения**: `OPEN_MAX` — максимальное количество дескрипторов, которые могут быть одновременно открыты некоторым процессом (POSIX устанавливает для этой величины ограничение снизу); `PIPE_BUF` — максимальное количество данных, для которого гарантируется атомарность операции записи (POSIX требует по менее 512 байт).

Значение `OPEN_MAX` можно узнать, вызвав функцию `sysconf`, его можно изменить из интерпретатора команд или из процесса. Значение `PIPE_BUF` обычно определено в заголовочном файле. Для FIFO с точки зрения стандарта POSIX оно представляет собой переменную (ее значение можно получить в момент выполнения программы), зависящую от полного имени файла, поскольку разные имена могут относиться к разным файловым системам, и эти файловые системы могут иметь различные характеристики.

4. Очереди сообщений

Очередь сообщений находится в адресном пространстве ядра и имеет ограниченный размер. В отличие от каналов, которые обладают теми же самыми свойствами, очереди сообщений сохраняют границы сообщений. Это значит, что ядро ОС гарантирует, что сообщение, поставленное в очередь, не смешается с предыдущим или следующим сообщением при чтении из очереди. Кроме того, с каждым сообщением связывается его тип. Процесс, читающий очередь сообщений, может отбирать только сообщения заданного типа или все сообщения кроме сообщений заданного типа.

Очередь сообщений можно рассматривать как связный список сообщений. Каждое сообщение представляет собой запись, очереди сообщений автоматически расставляют границы между записями, аналогично тому, как это делается в дейтаграммах UDP. Для записи сообщения в очередь не требуется наличия ожидающего его процесса в отличие от неименованных каналов и

FIFO, в которые нельзя произвести запись, пока не появится считывающий данные процесс. Поэтому процесс может записать в очередь какие-то сообщения, после чего они могут быть получены другим процессом в любое время, даже если первый завершит свою работу. С завершением процесса-источника данные не исчезают (данные, остающиеся в именованном или неименованном канале, сбрасываются, после того как все процессы закроют его).

Следует заметить, что, к сожалению, не определены системные вызовы, которые позволяют читать сразу из нескольких очередей сообщений, или из очередей сообщений и файловых дескрипторов. Видимо, отчасти и поэтому очереди сообщений широко не используются.

Пример.

ТЗ: Создать клиент-серверное приложение, демонстрирующее передачу информации между процессами посредством очередей сообщений.

Аналогично предыдущему разделу программа включает 2 файла: серверный и клиентский. В общем случае одновременно могут работать несколько клиентов.

Серверный файл содержит:

- подключение библиотек (см. листинг ниже)
- обработчик сигнала SIGINT (с восстановлением диспозиции и удалением очереди сообщений системным вызовом `msgctl()` для корректного завершения сервера при получении сигнала SIGINT);
- основную программу со следующей структурой:

```
void main(void)
{
    Message msg_rcv; //принимаемое сообщение
    Message msg_snd; //посылаемое сообщение
    key_t key; //ключ, необходимый для создания очереди
    int length, n;
    signal(SIGINT, sig_hdlr);
    //получение ключа
    if((key = ftok("/home/your_path/test.txt", 'A')) < 0)
        //ftok - преобразует имя файла и идентификатор проекта в ключ для
        //системных вызовов (для работы с очередью)
        {
            printf("Server : can't receive a key\n");
            exit(-1);
        }
}
```

далее создается очередь сообщений, используя системный вызов

```
msgget(key, PERM | IPC_CREAT),
```

организовывается цикл ожидания сообщения и его чтение. Сервер в цикле читает сообщения из очереди (тип = 1) функцией `msgrcv()` и посылает на каждое сообщение ответ клиенту (тип = 2) функцией `msgsnd()`. Целесообразно дублировать вывод сообщений на экран для контроля. В случае возникновения любых ошибок функцией `kill()` инициируется посылка сигнала SIGINT. Обработчик сигнала выполняет восстановление диспозиции сигналов и удаление очереди сообщений системным вызовом `msgctl()`.

В файле **client.c** аналогично серверному коду должен быть получен ключ, затем доступ к очереди сообщений, отправка сообщения серверу (тип 1). Затем организовывается цикл ожидания сообщения клиентом с последующим чтением (тип 2).

Таким образом, функции чтения и отправки сообщения реализуются системными вызовами: `msgrcv()`, `msgsnd()`.

Алгоритм работы программы:

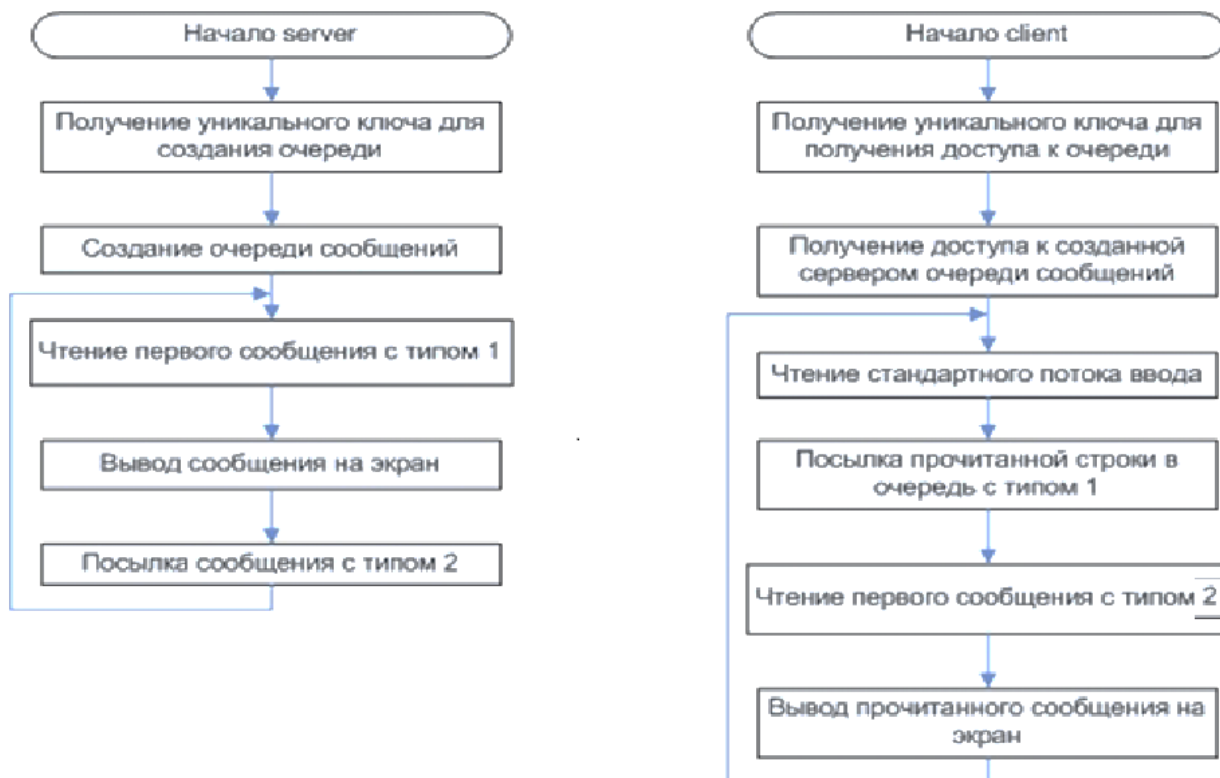


Рис.3. Взаимодействие посредством именованных каналов

Исходный код: server.c

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/types.h>
#include <signal.h>
#define DEF_KEY_FILE "key"
  
```

```

typedef struct {
    long type;
    char buf[100];
} Message;

int queue;
void intHandler(int sig) {
    signal(sig,SIG_DFL);
    if(msgctl(queue,IPC_RMID,0) < 0) {
        printf("Can't delete queue\n");
        exit(1);
    }
}

int main(int argc, char** argv) {
    char keyFile[100];
    bzero(keyFile,100);
    if(argc < 2) {
        printf("Using default key file %s\n",DEF_KEY_FILE);
        strcpy(keyFile,DEF_KEY_FILE);
    }
    else
        strcpy(keyFile,argv[1]);

    key_t key;
    key = ftok(keyFile,'Q');
    if(key == -1) {
        printf("no got key for the key file %s and id 'Q'\n",keyFile);
        exit(1);
    }

    queue = msgget(key,IPC_CREAT | 0666);
    if (queue < 0) {
        printf("Can't create queue\n");
        exit(4);
    }
    // до этого момента вызывали exit(), а не kill, т.к. очередь
    // еще не была создана
    signal(SIGINT,intHandler);
    // основной цикл работы сервера
    Message mes;
    int res;
    for(;;) {
        bzero(mes.buf,100);
        // получаем первое сообщение с типом 1
        res = msgrcv(queue,&mes,sizeof(Message),1L,0);
        if(res < 0) {
            printf("Error while recving msg\n");
            kill(getpid(),SIGINT);
        }
        printf("Client's request: %s\n",mes.buf);
        // шлем клиенту сообщение с типом 2, что все ок
        mes.type = 2L;
    }
}

```

```

        bzero(mes.buf,100);
        strcpy(mes.buf,"OK");
        res = msgsnd(queue,(void*)&mes,sizeof(Message),0);
        if(res != 0) {
            printf("error while sending msg\n");
            kill(getpid(),SIGINT);
        }
    }
    return 0;
}

```

Исходный код программы **client.c**:

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/types.h>
#include <signal.h>
#define DEF_KEY_FILE "key"
typedef struct {
    long type;
    char buf[100];
} Message;

int queue;

int main(int argc, char** argv) {
    char keyFile[100];
    bzero(keyFile,100);
    if(argc < 2) {
        printf("Using default key file %s\n",DEF_KEY_FILE);
        strcpy(keyFile,DEF_KEY_FILE);
    }
    else
        strcpy(keyFile,argv[1]);

    key_t key;
    key = ftok(keyFile,'Q');
    if(key == -1) {
        printf("no got key for key file %s and id 'Q'\n",keyFile);
        exit(1);
    }
    queue = msgget(key,0);
    if (queue < 0) {
        printf("Can't create queue\n");
        exit(4);
    }
    // основной цикл работы программы
    Message mes;
    int res;
    for(;;) {
        bzero(mes.buf,100);
        // читаем сообщение с консоли

```

```
fgets(mes.buf,100,stdin);
mes.buf[strlen(mes.buf) - 1] = '\\0';
// шлем его серверу
mes.type = 1L;
res = msgsnd(queue,(void*)&mes,sizeof(Message),0);
if(res != 0) {
    printf("Error while sending msg\\n");
    exit(1);
}
// получаем ответ, что все хорошо
res = msgrcv(queue,&mes,sizeof(Message),2L,0);
if(res < 0) {
    printf("Error while recving msg\\n");
    exit(1);
}
printf("Server's response: %s\\n",mes.buf);
}
return 0;
}
```

Описание работы сервера: Сервер получает ключ, по имени файла. С помощью ключа и идентификатора = 'Q' получает очередь сообщений и ждет сообщений с типом 1 от клиентов. При получении сообщения сервер выводит его на экран и отправляет обратное сообщение с типом 2, содержащее фразу «ОК».

Описание работы клиента: Клиент получает ту же очередь, что и сервер и ждет ввода пользователя. Считав ввод, он шлет сообщение с типом 1, содержащее считанные данные и ожидает от сервера подтверждения о принятии.

Запустим два процесса *client* и серверный процесс с разных терминалов.

Результаты выполнения

Сервер:

```
de@de:~/lab4/messages$ ./server
Using default key file key
Client's request: first client
Client's request: second client
```

Первый клиент:

```
de@de:~/lab4/messages$ ./client
Using default key file key
first client
Server's response: OK
```

Второй клиент:

```
de@de:~/lab4/messages$ ./client
Using default key file key
second client
Server's response: OK
```

Количественные ограничения средств ИРС

Максимальные и минимальные значения констант можно выяснить различными способами, в частности, просматривая соответствующие файлы каталога `/proc/sys/kernel`.

Наиболее простой способ – воспользоваться утилитой `ipcs` с ключом `-l`.

Пример выполнения команды `ipcs`:

```
de@de:/lab4/6-shm2s$ ipcs -l
----- Shared Memory Limits -----
max number of segments = 4096
max seg size (kbytes) = 32768
max total shared memory (kbytes) = 8388608
min seg size (bytes) = 1

----- Semaphore Limits -----
max number of arrays = 128
max semaphores per array = 250
max semaphores system wide = 32000
max ops per semop call = 32
semaphore max value = 32767

----- Messages Limits -----
max queues system wide= 1685
max size of message (bytes) = 8192
default max size of queue (bytes) = 16384
```

Например, если необходимо определить *максимальные* размер сообщения и одномоментное количество сообщений в очереди, то, как видим по результатам вывода утилиты, по умолчанию размер одного сообщения не может быть больше 8192 байт, а очередь не может содержать больше 1685 сообщений в один момент времени.

5. Семафоры и разделяемая память

Рассмотрим несколько вариантов постановки задачи синхронизации доступа к разделяемой памяти.

5.1. Вариант 1. Есть один процесс, выполняющий запись в разделяемую память и один процесс, выполняющий чтение из нее. Под чтением понимается извлечение данных из памяти. Программа должна обеспечить невозможность повторного чтения одних и тех же данных и невозможность перезаписи

данных, т.е. новой записи, до тех пор, пока читатель не прочитает предыдущую.

В таком варианте задания для синхронизации процессов достаточно двух семафоров.

Покажем, почему не достаточно одного **на примере**.

Так как мы используем один семафор, то алгоритм работы читателя и писателя может быть только таким – захват семафора, выполнение действия (чтение / запись), освобождение семафора. Теперь допустим, что читатель прочитал данные, освободил семафор и еще не до конца использовал квант процессорного времени. Тогда он перейдет на новую итерацию, снова захватит только что освобожденный семафор и снова прочитает данные – ошибка.

Теперь покажем, почему достаточно двух семафоров. Придадим одному из них смысл «запись разрешена», т.е. читатель предыдущие данные уже использовал; второму – «чтение разрешено», т.е. писатель уже сгенерировал новые данные, которые нужно прочитать.

Тогда **алгоритм** работы читателя и писателя будет выглядеть так:



Рис.4. Организация доступа к разделяемой памяти одного «читателя» и одного «писателя»

Оба семафора бинарные и используют стандартные операции, захват семафора – это ожидание освобождения ресурса (установки семафора в 1) и последующий захват ресурса (установки семафора в 0), освобождение ресурса – это установка семафора в 1.

Пару семафоров, использованных таким образом, иногда называют *разделенным бинарным семафором*, поскольку в любой момент времени только один из них может иметь значение 1.

При таком алгоритме работы, оба процесса после выполнения своей задачи и

освобождения одного из семафоров, будут ждать освобождения другого семафора, которое произведет другой процесс, но только после выполнения своей работы. Таким образом повторное чтение, или повторная запись стала невозможной.

Исходный код программы:

Процесс-читатель:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <sys/time.h>
#include "shm.h"

Message* p_msg;
int shmemory;
int semaphore;
void intHandler(int sig) {
    //отключаем разделяемую память
    if(shmdt(p_msg) < 0) {
        printf("Error while detaching shm\n");
        exit(1);
    }
    //удаляем shm и семафоры
    if(shmctl(shmemory, IPC_RMID, 0) < 0) {
        printf("Error while deleting shm\n");
        exit(1);
    }
    if(semctl(semaphore, 0, IPC_RMID) < 0) {
        printf("Error while deleting semaphore\n");
        exit(1);
    }
}
int main(int argc, char** argv) {
    char keyFile[100];
    bzero(keyFile,100);
    if(argc < 2) {
        printf("Using default key file %s\n",DEF_KEY_FILE);
        strcpy(keyFile,DEF_KEY_FILE);
    }
    else
        strcpy(keyFile,argv[1]);
    key_t key;

    //будем использовать 1 и тот же ключ для семафора и для shm
    if((key = ftok(keyFile, 'Q')) < 0) {
        printf("Can't get key for key file %s and id 'Q'\n",keyFile);
```

```

    exit(1);
}
//создаем shm
if((shmemory = shmget(key, sizeof(Message), IPC_CREAT | 0666)) < 0) {
    printf("Can't create shm\n");
    exit(1);
}
//присоединяем shm в наше адресное пространство
if((p_msg = (Message*)shmat(shmemory, 0, 0)) < 0) {
    printf("Error while attaching shm\n");
    exit(1);
}

    // устанавливаем обработчик сигнала
signal(SIGINT, intHandler);

//создаем группу из 2 семафоров
//1 - показывает, что можно читать
//2 - показывает, что можно писать
if((semaphore = semget(key, 2, IPC_CREAT | 0666)) < 0) {
    printf("Error while creating semaphore\n");
    kill(getpid(),SIGINT);
}
// устанавливаем 2 семафор в 1, т.е. можно писать
if(semop(semaphore, setWriteEna, 1) < 0) {
    printf("execution complete\n");
    kill(getpid(),SIGINT);
}
// основной цикл работы
for(;;) {
    // ждем пока клиент начнет работу
    if(semop(semaphore, readEna, 1) < 0) {
        printf("execution complete\n");
        kill(getpid(),SIGINT);
    }
    //читаем сообщение от клиента
    printf("Client's message: %s", p_msg->buf);
    // говорим клиенту, что можно снова писать
    if(semop(semaphore, setWriteEna, 1) < 0) {
        printf("execution complete\n");
        kill(getpid(),SIGINT);
    }
}
}
}

```

Процесс-писатель:

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include "shm.h"

```

```

int main(int argc, char** argv) {
    Message* p_msg;
    char keyFile[100];
    bzero(keyFile,100);
    if(argc < 2) {
        printf("Using default key file %s\n",DEF_KEY_FILE);
        strcpy(keyFile,DEF_KEY_FILE);
    }
    else
        strcpy(keyFile,argv[1]);
    key_t key;
int shmemory;
int semaphore;
//будем использовать 1 и тот же ключ для семафора и для shm
if((key = ftok(keyFile, 'Q')) < 0) {
    printf("Can't get key for key file %s and id 'Q'\n",keyFile);
    exit(1);
}
//создаем shm
if((shmemory = shmget(key, sizeof(Message), 0666)) < 0) {
    printf("Can't create shm\n");
    exit(1);
}
//присоединяем shm в наше адресное пространство
if((p_msg = (Message*)shmat(shmemory, 0, 0)) < 0) {
    printf("Error while attaching shm\n");
    exit(1);
}

if((semaphore = semget(key, 2, 0666)) < 0) {
    printf("Error while creating semaphore\n");
    exit(1);
}

    char buf[100];
    for(;;) {
        bzero(buf,100);
        printf("Type message to serever. Empty string to finish\n");
        fgets(buf,100,stdin);
        if(strlen(buf) == 1 && buf[0] == '\n') {
            printf("bye-bye\n");
            exit(0);
        }
        //хотим отправить сообщение
        if(semop(semaphore, writeEna, 1) < 0) {
            printf("Can't execute a operation\n");
            exit(1);
        }
        //запись сообщения в разделяемую память
        sprintf(p_msg->buf,"%s", buf);
        //говорим серверу, что он может читать
        if(semop(semaphore, setReadEna, 1) < 0) {
            printf("Can't execute a operation\n");
            exit(1);
        }
    }
}

```

```

}
}

//отключение от области разделяемой памяти
if(shmdt(p_msg) < 0) {
    printf("Error while detaching shm\n");
    exit(1);
}
}
}

```

Файл shm.h

```

#define DEF_KEY_FILE "key"
typedef struct {
    long type;
    char buf[100];
} Message;
static struct sembuf readEna[1] = {0,-1,0};
static struct sembuf writeEna[1] = {1,-1,0};

static struct sembuf setReadEna[1] = {0,1,0};
static struct sembuf setWriteEna[1] = {1,1,0};

```

Результат работы

Поток-писатель 1:

```

de@de:/lab4/6-shm$ ./client
Using default key file key
Type message to serever. Empty string to finish
hi
Type message to serever. Empty string to finish
first
Type message to serever. Empty string to finish
client
Type message to serever. Empty string to finish

bye-bye
de@de:/media/data/lab4/6-shm$

```

Поток-писатель 2:

```

de@de:/lab4/6-shm$ ./client

Using default key file key
Type message to serever. Empty string to finish
hey
Type message to serever. Empty string to finish
second
Type message to serever. Empty string to finish
client
Type message to serever. Empty string to finish

bye-bye
dm@dm-Lenovo-B550:/media/data/lab4/6-shm$

```

Поток-читатель:

```

de@de:/lab4/6-shm$ ./server
Using default key file key
Client's message: hi

```

```
Client's message: first
Client's message: client
Client's message: hey
Client's message: second
Client's message: client
^Cexecution complete
```

Все сообщения от клиента сервером прочитаны.

5.2. Вариант 2. К условиям предыдущей задачи добавляется условие корректной работы нескольких читателей и нескольких писателей одновременно. Как и в предыдущем варианте под чтением понимается извлечение данных из памяти, т. е. одну порцию данных может прочитать только один читатель.

Легко понять, что это условие не приводит к необходимости использования дополнительных средств синхронизации. Теперь вместо одного процесса, за каждый семафор будут конкурировать несколько. Но повторная запись или чтение все также невозможно. Так как, чтобы очередной процесс-писатель отработал, нужно освобождение семафора, которое выполняется из процесса-читателя, и наоборот.

5.3. Вариант 3. К условиям предыдущей задачи добавляется наличие не единичного буфера, а буфера некоторого размера. Тип буфера (очередь, стек, кольцевой буфер) не имеет значения.

Двух семафоров по прежнему достаточно, но это приведет к вырождению буфера, так как все процессы будут работать только с одной ячейкой.

Так как размер буфера не равен единице, то больше нет необходимости в чередовании операций чтения и записи, допустима ситуация нескольких записей подряд, и после этого нескольких чтений. Нужно только следить, чтобы не было записи в уже заполненный буфер и не было чтения из пустого буфера. Для этого выберем другие типы семафора и придадим им другую семантику. Возьмем два считающих семафора. Максимальное значение обоих – размер буфера. Первый инициализируется нулем и имеет смысл «количество заполненных ячеек», второй инициализируется N , где N – размер буфера и имеет смысл «количество пустых ячеек». Процессы-читатели перед своей работой захватывают семафор «количество заполненных ячеек», т.е. ждут появления хотя бы одной порции данных, а после чтения освобождают семафор «количество пустых ячеек». Процессы-писатели перед записью захватывают семафор «количество пустых ячеек», т.е. ждут появления хотя бы одной пустой ячейки для записи, а после записи освобождают семафор «количество полных

ячеек». Таким образом, решается проблема чтения из пустого буфера и запись в полный.

Так как семафоры не бинарные, захватить их может сразу несколько процессов, т.е. несколько процессов попадут в секцию записи или чтения. В этом случае, если операция записи или чтения не атомарная (а зачастую так оно и есть), может произойти нарушение нормальной работы программы, к примеру, несколько процессов-писателей попытаются произвести запись в одну и ту же ячейку буфера, или несколько читателей выполнят чтение одной и той же ячейки. Таким образом, операции записи-чтения становятся критическими секциями, доступ к которым также необходимо синхронизировать. Для этого будет достаточно еще одного бинарного семафора, имеющего смысл «доступ к памяти разрешен». Оба типа процессов должны захватывать его при попытке взаимодействия с памятью и освобождать после.

В итоге, **алгоритм** работы процессов-писателей, и процессов-читателей выглядит следующим образом:



Рис.5. Организация доступа к разделяемой памяти многих «читателей» и «писателей»

Важно отметить, что *порядок операций освобождения семафоров не важен, в то же время изменение порядка захвата семафоров может привести*

к взаимной блокировке процессов (dead lock). Взаимная блокировка в частности может произойти в таком случае: процесс-читатель захватил семафор «доступ к памяти разрешен», далее он проверяет, есть ли заполненные ячейки, т.е. пытается захватить семафор «количество заполненных ячеек», предположим, что свободных ячеек не оказалось и процесс переключился в неактивный режим, ожидая пока какой-нибудь процесс-писатель не запишет данные и не освободит семафор «количество заполненных ячеек». Но, этого никогда не произойдет, так как перед записью данных, процесс-писатель должен захватить семафор «доступ к памяти разрешен», который уже захвачен ожидающим процессом-читателем.

Пример решения последнего варианта задачи. В качестве разделяемого ресурса используется массив, находящийся в разделяемой памяти. Ячейка памяти, расположенная за последним элементом массива, интерпретируется как индекс последнего записанного элемента.

Исходный код программы:

Поток-читатель:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <sys/time.h>
#include "shm.h"

int* buf;
int shmemory;
int semaphore;
void intHandler(int sig) {
    shmdt(buf);
    shmctl(shmemory, IPC_RMID, 0);
    semctl(semaphore, 0, IPC_RMID);
}
int main(int argc, char** argv) {
    char keyFile[100];
    bzero(keyFile,100);
    if(argc < 2) {
        printf("Using default key file %s\n",DEF_KEY_FILE);
        strcpy(keyFile,DEF_KEY_FILE);
    }
    else
        strcpy(keyFile,argv[1]);
    key_t key;
```



```

//будем использовать 1 и тот же ключ для семафора и для shm
if((key = ftok(keyFile, 'Q')) < 0) {
    printf("Can't get key for key file %s and id 'Q'\n",keyFile);
    exit(1);
}
//создаем shm
if((shmemory = shmget(key,(BUF_SIZE+1)*sizeof(int),IPC_CREAT|0666))< 0)
{
    printf("Can't create shm\n");
    exit(1);
}
//присоединяем shm в наше адресное пространство
if((buf = (int*)shmat(shmemory, 0, 0)) < 0) {
    printf("Error while attaching shm\n");
    exit(1);
}

    // устанавливаем обработчик сигнала
signal(SIGINT, intHandler);

//создаем группу из 3 семафоров
//1 - число свободных ячеек
//2 - число занятых ячеек
// 3 работа с памятью
if((semaphore = semget(key, 3, IPC_CREAT | 0666)) < 0) {
    printf("Error while creating semaphore\n");
    kill(getpid(),SIGINT);
}
// устанавливаем индекс в -1,
//первый записывающий клиент установит его в ноль
buf[BUF_SIZE] = -1;
// инициализируем массив -1
int j = 0;
for(j = 0; j < BUF_SIZE; ++j) {
    buf[j] = -1;
}
//устанавливаем 1 семафор в число свободных ячеек, т.е. можно писать
if(semop(semaphore, setFree, 1) < 0) {
    printf("execution complete\n");
    kill(getpid(),SIGINT);
}
// говорим, что память свободна
if(semop(semaphore, mem_unlock, 1) < 0) {
    printf("execution complete\n");
    kill(getpid(),SIGINT);
}
printf("Press enter to start working\n");
getchar();
// основной цикл работы
int i = 0;
for(i = 0; i < 15; ++i) {
    // ждем, пока будет хоть одна непустая ячейка

```

```

    if(semop(semaphore, waitNotEmpty, 1) < 0) {
        printf("execution complete\n");
        kill(getpid(),SIGINT);
    }
    // ждем возможности поработать с памятью
    if(semop(semaphore, mem_lock, 1) < 0) {
        printf("execution complete\n");
        kill(getpid(),SIGINT);
    }

    //читаем сообщение от клиента
    int res = buf[buf[BUF_SIZE]];
    buf[BUF_SIZE] = buf[BUF_SIZE] - 1;
    printf("Remove %d from cell %d\n", res,buf[BUF_SIZE]+1);
    // освобождаем память
    if(semop(semaphore, mem_unlock, 1) < 0) {
        printf("execution complete\n");
        kill(getpid(),SIGINT);
    }
    // увеличиваем число пустых ячеек
    if(semop(semaphore, releaseEmpty, 1) < 0) {
        printf("execution complete\n");
        kill(getpid(),SIGINT);
    }
}
}
}

```

Поток-писатель:

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include "shm.h"

int* buf;

int main(int argc, char** argv) {
    char keyFile[100];
    bzero(keyFile,100);
    if(argc < 2) {
        printf("Using default key file %s\n",DEF_KEY_FILE);
        strcpy(keyFile,DEF_KEY_FILE);
    }
    else
        strcpy(keyFile,argv[1]);
    key_t key;
    int shmemory;
    int semaphore;
    //будем использовать 1 и тот же ключ для семафора и для shm
    if((key = ftok(keyFile, 'Q')) < 0) {
        printf("Can't get key for key file %s and id 'Q'\n",keyFile);
        exit(1);
    }
}

```

```

}
//создаем shm
if((shmemory = shmget(key, (BUF_SIZE+1)*sizeof(int), 0666)) < 0)
{
    printf("Can't create shm\n");
    exit(1);
}
//присоединяем shm в наше адресное пространство
if((buf = (int*)shmat(shmemory, 0, 0)) < 0) {
    printf("Error while attaching shm\n");
    exit(1);
}

if((semaphore = semget(key, 2, 0666)) < 0) {
    printf("Error while creating semaphore\n");
    exit(1);
}
printf("Press enter to start working\n");
getchar();
int send = 0;
char tb[10];
int i = 0;
    for(i = 0; i < 10;++i) {
//ждем, пока будет хоть одна свободная ячейка
if(semop(semaphore, waitNotFull, 1) < 0) {
    printf("Can't execute a operation\n");
    exit(1);
}
// ждем доступа к разделяемой памяти
if(semop(semaphore, mem_lock, 1) < 0) {
    printf("Can't execute a operation\n");
    exit(1);
}
printf("Add %d to cell %d\n",send,buf[BUF_SIZE]+1);
    ++buf[BUF_SIZE];
buf[buf[BUF_SIZE]] = send++;
//освобождаем доступ к памяти
if(semop(semaphore, mem_unlock, 1) < 0) {
    printf("Can't execute a operation\n");
    exit(11);
}
//увеличиваем число занятых ячеек
if(semop(semaphore, releaseFull, 1) < 0) {
    printf("Can't execute a operation\n");
    exit(11);
}
}
//отключение от области разделяемой памяти
shmdt(buf);
}

```

Результаты выполнения

Процесс-писатель 1:

```
de@de:/lab4/9-shmdop$ ./client
Using default key file key
Press enter to start working

Add 0 to cell 0
Add 1 to cell 1
Add 2 to cell 2
Add 3 to cell 3
Add 4 to cell 4
Add 5 to cell 0
Add 6 to cell 1
Add 7 to cell 2
Add 8 to cell 3
Add 9 to cell 4
```

Процесс-писатель 2:

```
de@de:/lab4/9-shmdop$ ./client
Using default key file key
Press enter to start working

Add 0 to cell 0
Add 1 to cell 1
Add 2 to cell 2
Add 3 to cell 3
Add 4 to cell 4
Add 5 to cell 0
Add 6 to cell 1
Add 7 to cell 2
Add 8 to cell 3
Add 9 to cell 4
```

Процесс-читатель:

```
de@de:/lab4/9-shmdop$ ./server
Using default key file key
Press enter to start working

Remove 4 from cell 4
Remove 3 from cell 3
Remove 2 from cell 2
Remove 1 from cell 1
Remove 0 from cell 0
Remove 9 from cell 4
Remove 8 from cell 3
Remove 7 from cell 2
Remove 6 from cell 1
Remove 5 from cell 0
Remove 4 from cell 4
Remove 3 from cell 3
Remove 2 from cell 2
Remove 1 from cell 1
Remove 0 from cell 0
```

Процессы-писатели записывают по 10 чисел в массив, процесс-читатель считывает первые 15 из записанных. По результатам проверяем, что синхронизация работает корректно, выхода за пределы массива нет, записанные данные не затираются до их прочтения. К примеру, в нулевую ячейку сначала первый поток-писатель записал 0, потом он же записал 5, а потом второй поток-

писатель записал также 0. Проверяем, что все данные были прочитаны.

6. Сокеты

Пример использования сокетов — простейший эхо-сервер.

ТЗ: Сервер прослушивает заданный порт, при запросе нового соединения, создается новый поток для его обработки. Работа с клиентом должна быть организована как бесконечный цикл, в котором выполняется прием сообщения от клиента, вывод его на экран и пересылка обратно клиенту.

Клиентская программа после установления соединения с сервером также в бесконечном цикле выполняет чтение ввода пользователя, пересылку серверу, и получение работы. Если была введена пустая строка, клиент завершается.

Алгоритм работы программы:

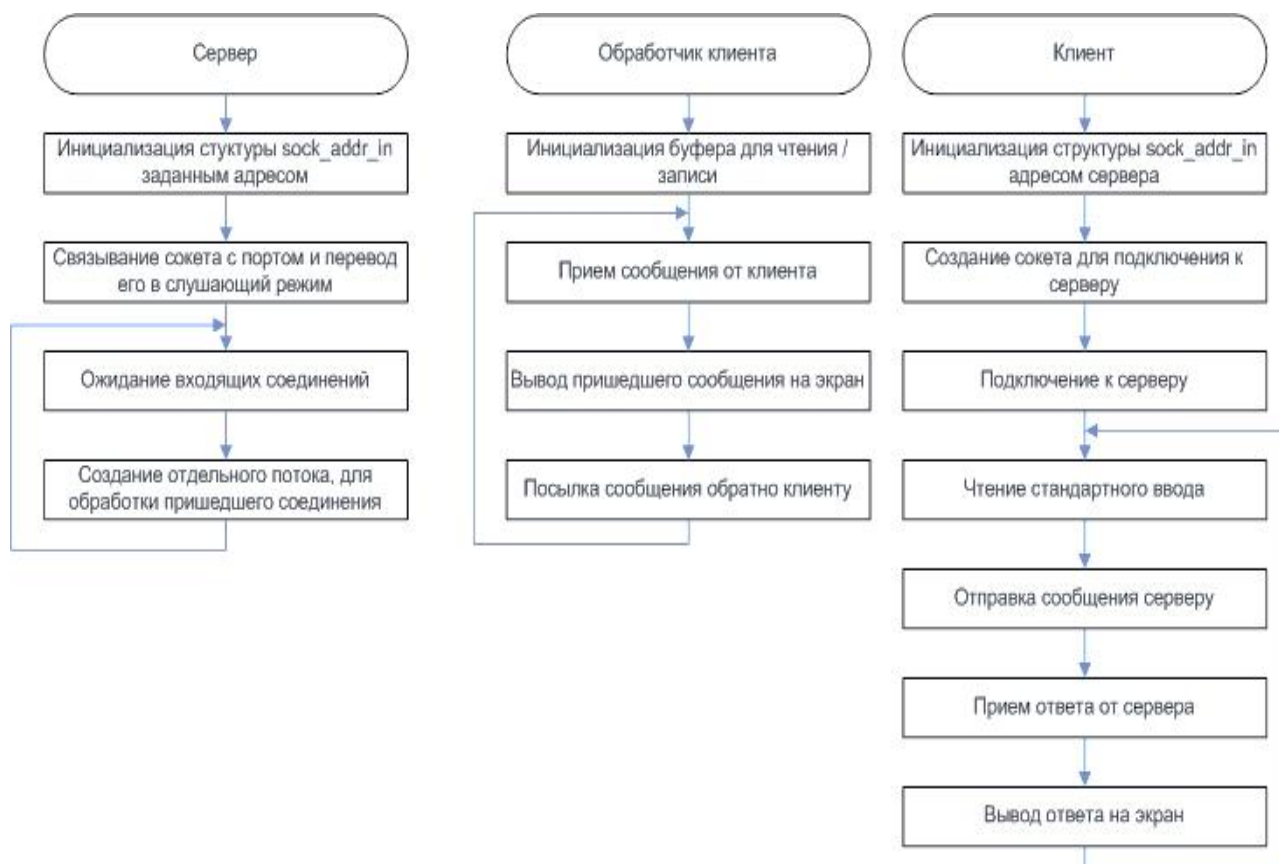


Рис. 6 Алгоритм простейшего эхо-сервера

Исходный код (файл `server.c`):

```
#include <sys/types.h>
#include <sys/socket.h>
```

```

#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define DEF_PORT 8888
#define DEF_IP "127.0.0.1"

// обработка одного клиента
void* clientHandler(void* args) {
    int sock = (int)args;
    char buf[100];
    int res = 0;
    for(;;) {
        bzero(buf,100);
        res = readFix(sock, buf,100, 0);
        if ( res <= 0 ) {
            perror( "Can't recv data from client, ending thread\n" );
            pthread_exit(NULL);
        }
        printf( "Some client sent: %s\n",buf);
        res = sendFix(sock,buf,0);
        if ( res <= 0 ) {
            perror( "send call failed" );
            pthread_exit(NULL);
        }
    }
}

int main( int argc, char** argv) {
    int port = 0;
    if(argc < 2) {
        printf("Using default port %d\n",DEF_PORT);
        port = DEF_PORT;
    } else
        port = atoi(argv[1]);
    struct sockaddr_in listenerInfo;
    listenerInfo.sin_family = AF_INET;
    listenerInfo.sin_port = htons( port );
    listenerInfo.sin_addr.s_addr = htonl( INADDR_ANY );
    int listener = socket(AF_INET, SOCK_STREAM, 0 );
    if ( listener < 0 ) {
        perror( "Can't create socket to listen: " );
        exit(1);
    }
    int res = bind(listener,(struct sockaddr *)&listenerInfo, sizeof(listenerInfo));
    if ( res < 0 ) {
        perror( "Can't bind socket" );
        exit( 1 );
    }
    // слушаем входящие соединения
    res = listen(listener,5);

```

```

    if (res) {
        perror("Error while listening:");
        exit(1);
    }
    // основной цикл работы
    for(;;) {
        int client = accept(listener, NULL, NULL );
        pthread_t thrd;
        res = pthread_create(&thrd, NULL, clientHandler, (void
*)(client));
        if (res){
            printf("Error while creating new thread\n");
        }
    }
    return 0;
}

int readFix(int sock, char* buf, int bufSize, int flags) {
    // читаем "заголовок" - сколько байт составляет наше сообщение
    unsigned msgLength = 0;
    int res=recv(sock,&msgLength,sizeof(unsigned),flags|MSG_WAITALL );
    if (res <= 0)return res;
    if(res > bufSize) {
        printf("Recieved more data, then we can store, exiting\n");
        exit(1);
    }
    // читаем само сообщение
    return recv(sock, buf, msgLength, flags | MSG_WAITALL);
}

int sendFix(int sock, char* buf, int flags) {
    // шлем число байт в сообщении
    unsigned msgLength = strlen(buf);
    int res = send(sock, &msgLength, sizeof(unsigned), flags );
    if (res <= 0)
        return res;
    send(sock, buf, msgLength, flags);
}

```

Исходный код (файл client.c):

```

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define DEF_PORT 8888
#define DEF_IP "127.0.0.1"

int main( int argc, char** argv) {
    char* addr;

```

```

int port;
char* readbuf;
if(argc <3) {
    printf("Using default port %d\n",DEF_PORT);
    port = DEF_PORT;
} else
    port = atoi(argv[2]);

if(argc < 2) {
    printf("Using default addr %s\n",DEF_IP);
    addr = DEF_IP;
} else
    addr = argv[1];

// создаем сокет
struct sockaddr_in peer;
peer.sin_family = AF_INET;
peer.sin_port = htons( port );
peer.sin_addr.s_addr = inet_addr( addr );
int sock = socket( AF_INET, SOCK_STREAM, 0 );
if ( sock < 0 ){
    perror( "Can't create socket\n" );
    exit( 1 );
}
// присоединяемся к серверу
int res = connect( sock, ( struct sockaddr * )&peer, sizeof(
peer ) );
if (res) {
    perror( "Can't connect to server:" );
    exit( 1 );
}
// основной цикл программы
char buf[100];
for(;;) {
    printf("Input request (empty to exit)\n");
    bzero(buf,100);
    fgets(buf, 100, stdin);
    buf[strlen(buf)-1] = '\0';
    if(strlen(buf) == 0) {
        printf("Bye-bye\n");
        return 0;
    }
    res = sendFix(sock, buf,0);
    if ( res <= 0 ) {
        perror( "Error while sending:" );
        exit( 1 );
    }
    bzero(buf,100);
    res = readFix(sock, buf, 100, 0);
    if ( res <= 0 ) {
        perror( "Error while receiving:" );
        exit(1);
    }
}

```



```

        printf("Server's response: %s\n",buf);
    }
    return 0;
}

int readFix(int sock, char* buf, int bufSize, int flags) {
    // читаем "заголовок" - сколько байт составляет наше сообщение
    unsigned msgLength = 0;
    int res=recv(sock,&msgLength,sizeof(unsigned),flags|MSG_WAITALL);
    if (res <= 0)return res;
    if(res > bufSize) {
        printf("Recieved more data, then we can store, exiting\n");
        exit(1);
    }
    // читаем само сообщение
    return recv(sock, buf, msgLength, flags | MSG_WAITALL);
}

int sendFix(int sock, char* buf, int flags) {
    // число байт в сообщении
    unsigned msgLength = strlen(buf);
    int res = send(sock, &msgLength, sizeof(unsigned), flags );
    if (res <= 0)
        return res;
    send(sock, buf, msgLength, flags);
}

```

Для взаимодействия используются TCP сокет, это значит, что между сервером и клиентом устанавливается логическое соединение, при этом при получении данных из сокета с помощью вызова `recv`, есть вероятность получить сразу несколько сообщений, или не полностью прочитать сообщение. Поэтому для установления взаимной однозначности между отосланными и принятыми данными используются функции `recvFix` и `sendFix`. Принцип их работы следующий: функция `sendFix` перед посылкой собственно данных посылает «заголовок» - количество байт в посылке. Функция `recvFix` вначале принимает этот «заголовок», и вторым вызовом `recv` считывает переданное количество байт. Считать ровно то, количество байт, которое указано в аргументе функции `recv`, позволяет флаг `MSG_WAITALL`. Если его не использовать и данных в буфере недостаточно, то будет прочитано меньшее количество.

Результат выполнения клиент-серверного приложения

Клиент:

```

de@de:~/lab4/6-socket$ ./client
Using default port 8888
Using default addr 127.0.0.1
Input request (empty to exit)

```

```
hi
Server's response: hi
Input request (empty to exit)
it's client
Server's response: it's client
Input request (empty to exit)
bye
Server's response: bye
Input request (empty to exit)

Bye-bye
```

Сервер:

```
de@de:~/lab4/6-socket$ ./server
Using default port 8888
Some client sent: hi
Some client sent: it's client
Some client sent: bye
Can't recv data from client, ending thread
: Success
^C
de@de:~/lab4/6-socket$
```

Пример использования программы в локальной сети.

Для этого программу-сервер запустим на одном компьютере, программу-клиент – на другом. Перед запуском программы-клиента необходимо узнать адрес сервера с помощью команды **ifconfig**:

```
g4081_13@SPOComp8:~$ ifconfig
br0      Link encap:Ethernet  HWaddr 48:5b:39:78:43:3e
         inet addr:192.168.100.1  Bcast:192.168.100.255
Mask:255.255.255.0
         inet6 addr: fe80::4a5b:39ff:fe78:433e/64 Scope:Link
         UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
         RX packets:5 errors:0 dropped:0 overruns:0 frame:0
         TX packets:6 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:0
         RX bytes:264 (264.0 B)  TX bytes:468 (468.0 B)

eth0     Link encap:Ethernet  HWaddr 48:5b:39:78:43:3e
         inet addr:10.1.208.8  Bcast:10.1.208.31
Mask:255.255.255.224
         inet6 addr: fe80::4a5b:39ff:fe78:433e/64 Scope:Link
         UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
         RX packets:880 errors:0 dropped:0 overruns:0 frame:0
         TX packets:48 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1000
         RX bytes:67154 (67.1 KB)  TX bytes:4993 (4.9 KB)
         Interrupt:41
```

Таким образом, IP адрес сервера – 10.1.208.8

После запуска сервера и клиента убеждаемся, что программа работает

корректно:

Сервер:

```
de@de: /lab4/7-socket$ ./server
Using default port 8888
Some client sent: hi
Some client sent: it's me
Some client sent: how are y
Can't recv data from client, ending thread
```

Клиент:

```
de@de: /lab4/7-socket$ ./client 10.1.208.8
Using default port 8888
Input request (empty to exit)
hi
Server's response: hi
Input request (empty to exit)
it's me
Server's response: it's me
Input request (empty to exit)
how are y
Server's response: how are y
Input request (empty to exit)

Bye-bye
de@de:/lab4/7-socket$ ^C
de@de: /lab4/7-socket$
```

Самостоятельно

Модифицировать, если необходимо, предложенное приложение и реализовать обмен сервера с множеством клиентов. Количество клиентов: 10, 100, 1000.

Выполнить аналогичное взаимодействие на основе UDP, сравнить с IPC очереди сообщений.

Заключение по разделам ОС UNIX

В ОС Unix адресные пространства различных процессов изолированы друг от друга. Для взаимодействия процессов используются специальные средства IPC, включающие в себя сигналы, именованные и неименованные каналы, сообщения, сокеты, семафоры и разделяемую память.

В Unix поддерживается два вида сигналов: надёжные и ненадёжные. Ненадёжные сигналы более просты в использовании, в тоже время надёжные сигналы позволяют отложить прием других сигналов до окончания обработки текущего.

Сигналы самое простое средство IPC, являются достаточно медленными и ресурсоёмкими, не позволяют передавать произвольные данные, служат главным образом для уведомления, обработки нештатных ситуаций и синхронизации.

Именованные и неименованные каналы реализуют запись и чтение по принципу FIFO. Запись и чтение, таким образом, происходит быстро, однако при создании именованного канала затрачивается несколько больше времени. Кроме того, каналы работают в полудуплексном режиме, т.е. передают данные только в одну сторону. Каналы FIFO представляют собой вид IPC, который может использоваться только в пределах одного узла. Хотя FIFO и обладают именами в файловой системе, они могут применяться только в локальных файловых системах

Сообщения являются мощным средством межпроцессного обмена данными. Время доставки сообщения сравнимо с временем доставки сигнала, однако сообщение несёт гораздо больше информации, чем сигнал. С помощью сообщений гораздо проще организовать асинхронный обмен данными между процессами, чем с помощью каналов.

Семафоры и разделяемая память зачастую используются вместе. Семафоры позволяют синхронизировать доступ к разделяемому ресурсу и гарантировать «взаимное исключение» нескольких процессов при разделении ресурса (пока предыдущий процесс не закончит работу с ресурсом, следующий не начнет ее).

Сокеты являются средством IPC, которое можно использовать не только между процессами на одном компьютере, но и в сетевом режиме. Многие сетевые приложения построены на основе сокетов.

Системное программирование в ОС семейства Windows

Раздел 2. Управление процессами и потоками в Windows

Создание системных приложений в ОС Windows осуществляется с применением Application Programming Interface (интерфейса программирования приложений) (API).

Справочная информация для программиста MSDN доступна по ссылке на сайт производителя: <http://msdn.microsoft.com/library/>

Многочисленные функции API обеспечивают возможность доступа к системным ресурсам.

Перечислим основные функции управления процессами и потоками: создания, завершения, управления приоритетами

Функции управления процессами и потоками

<i>CreateProcess</i>	<i>CreateThread</i>
<i>GetPriorityClass,</i>	<i>GetThreadPriority,</i>
<i>SetPriorityClass,</i>	<i>SetThreadPriority</i>
<i>SetProcessPriorityBoost,</i>	<i>SetThreadPriorityBoost,</i>
<i>GetProcessPriorityBoost</i>	<i>ThreadPriorityBoost,</i>
<i>ExitProcess,</i>	<i>SuspendThread</i>
<i>TerminateProcess,</i>	<i>ResumeThread</i>

1. Создание процессов

Рассмотрим создание одного процесса другим посредством функции WinAPI *CreateProcess*. При создании исполнительная система выполняет работу по организации окружения (среды исполнения процесса) и предоставлению необходимых ему ресурсов. Она выделяет новое адресное пространство и иные ресурсы для процесса, а также создает для него новый базовый поток. Когда новый процесс будет 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 порядка десяти параметров, причем некоторые из них достаточно сложные и информационно емкие. После того, как исполнительная система создала новый процесс, она возвращает его описатель, а также описатель его базового потока.

Синтаксис команды *CreateProcess*.

```

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

```

Десять параметров *CreateProcess*() обеспечивают большую гибкость при использовании программистом, в простейшем случае для многих параметров можно использовать значения по умолчанию.

Параметры lpApplicationName u lpCommandLine

Используются вместе для указания исполняемой программы и аргументов командной строки.

Параметры lpProcessAttributes, lpThreadAttributes u bInheritHandles

Первые два – указатели на атрибуты безопасности для процесса и потока соответственно, последний параметр – флаг наследования (наследуются ли файловые дескрипторы и т.д.).

Параметр `DwCreationFlags`

Может объединять в себе несколько флаговых значений, включая следующие:

- `CREATE_SUSPENDED` — указывает на то, что основной поток будет создан в приостановленном состоянии и начнет выполняться лишь после вызова функция `ResumeThread`;

- `DETACHED_PROCESS` и `CREATE_NEW_CONSOLE` — взаимоисключающие значения, которые не должны устанавливаться оба одновременно. Первый флаг означает создание нового процесса, у которого консоль отсутствует, а второй — процесса, у которого имеется собственная консоль. Если ни один из этих флагов не указан, то новый процесс наследует консоль родительского процесса;

- `CREATE_NEW_PROCESS_GROUP` — указывает на то, что создаваемый процесс является корневым для новой группы процессов. Если все процессы, принадлежащие данной группе, разделяют общую консоль, то все они будут получать управляющие сигналы консоли (`Ctrl-C` или `Ctrl-break`);

В качестве флагов так же могут быть указаны приоритеты: `HIGH_PRIORITY_CLASS`, `IDLE_PRIORITY_CLASS`, `NORMAL_PRIORITY_CLASS` или `REALTIME_PRIORITY_CLASS`. Значение по умолчанию - `NORMAL_PRIORITY_CLASS`, но если порождающий процесс имеет приоритет `IDLE_PRIORITY_CLASS`, то и процесс-потомок также будет иметь приоритет `IDLE_PRIORITY_CLASS`.

Параметр `lpEnvironment`

Используется для передачи нового блока переменных окружения порожденному процессу-потомку. Если `NULL`, то потомок использует то же окружение, что и родитель. Если не `NULL`, то `lpEnvironment` должен указывать на массив строк, каждая `name=value`.

Параметр `lpCurrentDirectory`

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

Параметр `lpStartupInfo` – это указатель на следующую структуру:

```
typedef struct STARTUPINFO {
    DWORD cb; // длина структуры (обязательно инициализируем)
    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;
};

```

Экземпляр этой структуры данных должен быть создан в вызывающей программе. Затем ее адрес должен быть передан как параметр в `CreateProcess`.

Параметр `lpProcessInformation` – это указатель на структуру:

```

typedef struct _PROCESS_INFORMATION {
    HANDLE hProcess;    // описатель нового процесса
    HANDLE hThread;    // описатель потока нового процесса
    DWORD  dwProcessId; // глобальный идентификатор созданного процесса
    DWORD  dwThreadId;  // глобальный идентификатор потока созданного
процесса
};

```

`CreateProcess()` отводит место под объекты процесс и поток и возвращает значения их описателей (индексы в таблице) в структуре `PROCESS_INFORMATION`.

Освободить выделенное место можно вызовом `CloseHandle`. При этом выполнение этого вызова не обязательно приведет к завершению процесса (только исчезнет ссылка на объект внутри вызвавшего процесса).

`CreateProcess()` возвращает ноль, если создание процесса прошло успешно.

Примеры применения функций

Пример 1.1. Создание процесса для запуска приложения

Задание. Программа после запуска должна создать новый процесс, с помощью функции `CreateProcess`. В новом процессе необходимо запустить любое приложение (например, `notepad.exe` или `calc.exe`). Для контроля можно вывести идентификаторы созданного процесса и потока, а затем завершить основную программу.

Исходный код программы (task_1.cpp) , создающей процесс для запуска приложения notepad.exe и открытия блокнота с файлом без имени:

Листинг 1. Файл task_1.cpp

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

int main(int argc, char** argv[])
{
    printf("Program started\n");
    char LpCommandLine[60];
    strcpy_s(LpCommandLine, "C:\\WINDOWS\\system32\\notepad.exe");
    STARTUPINFO startupInfo;
    PROCESS_INFORMATION processInfo;          //информация о
процессе
    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))
    {
        printf("Error creating process: %d\n", GetLastError());
        return -1;
    }
    else
    {
        printf("new process Handle: %d Handle of thread: %d\n",
processInfo.dwProcessId,processInfo.dwThreadId);
        printf("Successfully created new process!\n");
    }
    CloseHandle(processInfo.hThread);
    CloseHandle(processInfo.hProcess);
    printf("Program finished\n");
    getchar();
    return 0;
}
```

Комментарии по значениям параметров функции *CreateProcess()*

Параметр *lpApplicationName* может быть (NULL). В этом случае, имя модуля должно быть в строке *lpCommandLine*.

В данном случае будем использовать значения по умолчанию для атрибутов безопасности процесса и потока – NULL (параметры *lpProcessAttributes* и *lpThreadAttributes*) и FALSE для флага наследования (*bInheritHandles*).

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

Параметр `lpEnvironment` используется для передачи нового блока переменных окружения порожденному процессу-потомку (`child`). Если указано `NULL` – то потомок использует то же окружение, что и родитель.

Параметр `lpCurrentDirectory` установлен в (`NULL`). Это означает, что новый процесс создается с тем же самым текущим диском и каталогом, что и вызывающий процесс.

В структуре `startupInfo` устанавливаются оконный режим терминала, рабочий стол, стандартные дескрипторы и внешний вид главного окна для нового процесса.

В структуре `processInfo` хранятся: описатель вновь созданного процесса (`hProcess`), описатель его базового потока (`hThread`), глобальный идентификатор процесса (`dwProcessId`), глобальный идентификатор потока (`dwThreadId`).

Результат выполнения программы представлен на рис. 7.

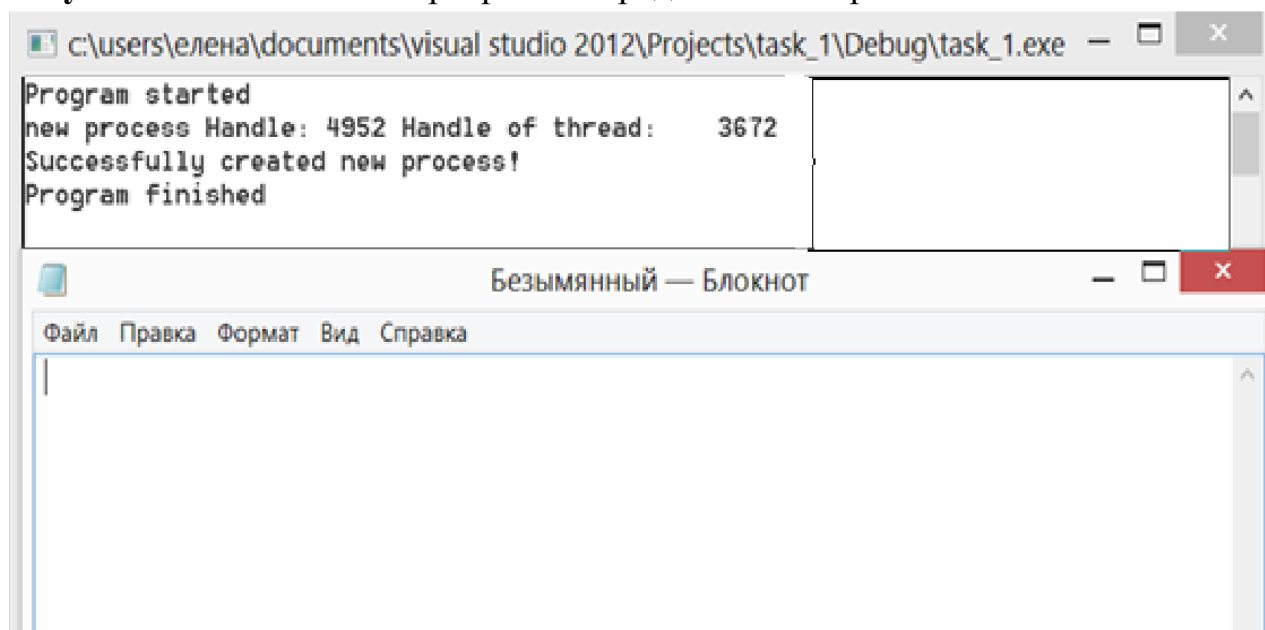


Рис. 7. Результат выполнения программы task_1.cpp

Пример 1.2. Создание процессов при работе с конфигурационным файлом

Задание. Программа, получает имя конфигурационного файла из командной строки, открывает конфигурационный файл, читает строки и создает для запуска каждой команды отдельный процесс.

С целью упрощения кода сначала имя командного файла зададим прямо в программе и представим текст программы без обработки возможных ошибок. Создаем конфигурационный файл с именем "temp.txt" при помощи текстового редактора (например, `notepad`) и располагаем его в корневом каталоге диска `C`. Далее записываем в файл следующие строки:

C:\Windows\System32\notepad.exe C:\temp.txt

C:\Windows\System32\calc.exe

Листинг 2. Файл task_2.cpp

```
// Примечания к коду (используем Microsoft Visual Studio 2012 и 2013)
// -----
//
// 1) меняем кодировку с юникода на многобайтовую:
// проект-> свойства -> общие -> набор символов -> использовать
// многобайтовую кодировку
// 2) перед всеми #include обязательно надо прописать #define
// _CRT_SECURE_NO_WARNINGS, т.к. код содержит устаревшие функции, которые
// не поддерживает MVS 2012
// 3) внимательно прописываем адрес, где лежит блокнот и калькулятор,
// а так же доступ к файлу (test.txt), который содержит команды
// -----
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <windows.h>
#include <assert.h>
#include <iostream>
#define MAX_LEN 200
using namespace std;
int main(int argc, char* argv[])
{

    const char* frd = "C:\\temp.txt";
    FILE *f=fopen(frd,"r");
    if(f==NULL) {
        cout<<"Coudn't open file"<<endl;
        system("pause");
        return 1;
    }

    for(int i=0; i<2; i++)
    {
        char* execString = (char*)calloc(MAX_LEN,
sizeof(char)); //выделение памяти
        fgets(execString, MAX_LEN, f); // чтение строки из файла
        execString[strlen(execString) - 1] = '\0';
        STARTUPINFO startupInfo;
        ZeroMemory(&startupInfo, sizeof(STARTUPINFO));
        startupInfo.cb = sizeof(startupInfo);
        PROCESS_INFORMATION processInfo;
        printf("%s\n", execString);
        if( !CreateProcess(NULL, execString, NULL, NULL, false, 0,
NULL, NULL, &startupInfo, &processInfo) )
        {
            printf("Error creating process: %d\n",
```

```

GetLastError());
        system("pause");
        return -1;
    }
    else printf("Process successfully created!\n");
    free(execString);
    CloseHandle(processInfo.hThread);
    CloseHandle(processInfo.hProcess);
}
system("pause");
return 0;
}

```

Результат выполнения программы представлен на рис. 8.

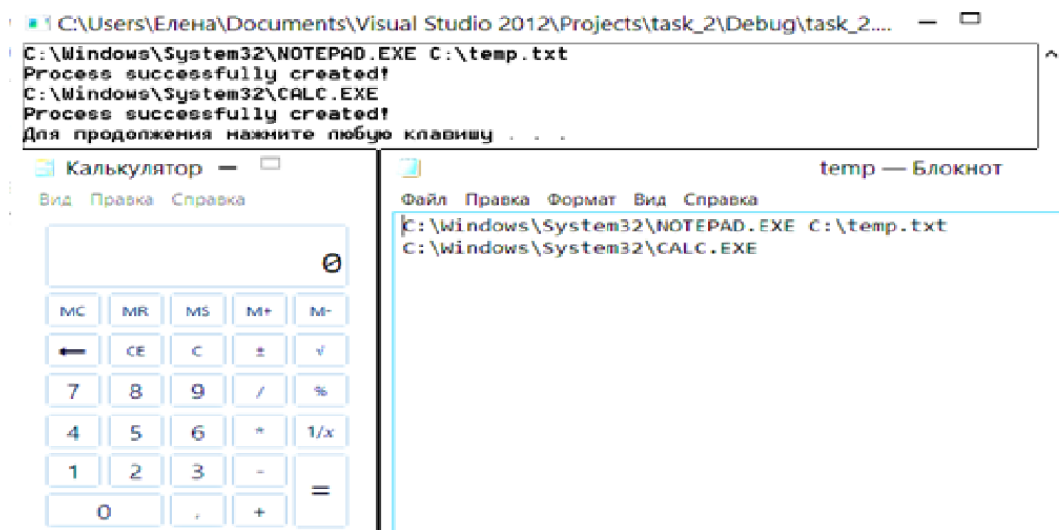


Рис. 8. Результат выполнения программы *task_2.cpp*

Пример 1.3. Доработаем программу. Пусть программа получает имя конфигурационного файла из командной строки, открывает его с помощью *fopen()*, читает построчно функцией *fgets()*. После прочтения каждой строки, если она не пуста, создается процесс, в командную строку которого пишется прочитанная строка. Если создать процесс не удалось, программа пробует читать конфигурационный файл дальше.

Листинг 3. Файл *task_2Change.cpp*

```

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <windows.h>
#include <assert.h>
#define DEF_BUFLLEN 100

int main(int argc, char* argv[]) {
    printf("Program started\n");
    if (argc < 2) {

```

```

        printf("Input name of configuration file\n");
        exit(999);
    }
    const char* frd = argv[1];
    FILE *f = fopen(frd, "r"); // открываем конфигурационный файл
//на чтение
    if(f==NULL) {
        printf("error opening file %s\n",argv[1]);
        exit(1000);
    }
    char commandLine[DEF_BUFLEN]; // буфер для читаемой строки
    STARTUPINFO StartupInfo;
    PROCESS_INFORMATION ProcessInformation;
    while (!feof(f)) {
        ZeroMemory(commandLine,DEF_BUFLEN);
        fgets(commandLine, DEF_BUFLEN, f);
        if(strlen(commandLine) <= 1) {
            printf("skipping empty string\n");
            continue;
        }
        commandLine[strlen(commandLine) - 1] = '\0';

        ZeroMemory(&StartupInfo, sizeof(STARTUPINFO));
        StartupInfo.cb = sizeof(STARTUPINFO);
        printf("Try to create new process with command line
%s'\n", commandLine);
        if( !CreateProcess(NULL, commandLine, NULL, NULL,false, 0,
NULL, NULL, &StartupInfo, &ProcessInformation) )
        {
            printf("Can't create new process.Error is:
%d\nContinue with next line\n", GetLastError());
            continue;
        }
        printf("new process Handle: %d Handle of thread:
%d\n",ProcessInformation.dwProcessId,ProcessInformation.dwThreadId);
        CloseHandle(ProcessInformation.hThread);
        CloseHandle(ProcessInformation.hProcess);
    }
    fclose(f);
    printf("Program finished\n");
    getchar();
    return 0;
}

```

Отметим, что вывод процессы производится здесь без синхронизации. Процесс-родитель не дожидается создания процессов-потомков, а им после создания необходимо разбирать строку аргументов. Когда передается один неверный аргумент, процесс не создается (т.к. сразу проверяется наличие

исполняемого файла с указанным именем). «Error 2» соответствует ошибка «Файл не найден». При передаче в `commandLine` процесса строки из нескольких слов процесс все равно создается (даже если аргументы не верны), но быстро завершается. При этом созданные процессы связаны с консолью процесса-родителя (аргумент `DwCreationFlags` в вызове `CreateProcess` равен 0).

Результат выполнения программы представлен на рис. 9.

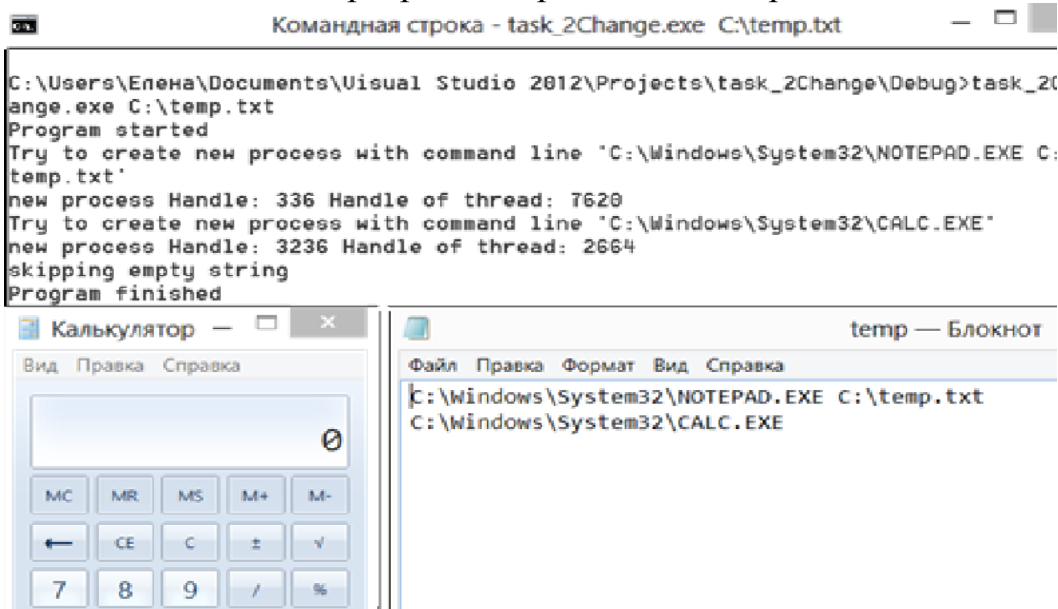


Рис. 9. Результат выполнения программы `task_2Change.cpp`

2. Создание потоков

Рассмотрим создание потоков посредством функции WinAPI `CreateThread`.

Синтаксис команды `CreateThread`:

```

HANDLE CreateThread(
LPSECURITY_ATTRIBUTES lpsa,           // дескриптор защиты
DWORD dwStackSize,                   // начальный размер стека
LPTHREAD_START_ROUTINE lpStartAddr,  // функция потока
LPVOID lpThreadParm,                // параметр потока
DWORD dwCreationFlags,               // опции создания
LPDWORD lpThreadId                   // идентификатор потока
)
  
```

lpsa — указатель на структуру с атрибутами защиты.

dwStackSize — размер стека нового потока в байтах. Значению 0 этого параметра соответствует размер стека по умолчанию, равный размеру стека основного потока.

lpStartAddr — указатель на функцию (принадлежащую контексту процесса), которая должна выполняться. Эта функция принимает единственный аргумент в виде указателя и возвращает 32-битовый код завершения. Этот аргумент может интерпретироваться потоком либо как переменная типа `DWORD`, либо как указатель.

Функция потока (`ThreadFunc`) имеет следующую сигнатуру:

lpThreadParm — параметр главной функции потока.

dwCreationFlags — если значение этого параметра установлено равным 0, то поток запускается сразу же после вызова функции `CreateThread`. Установка значения `CREATE_SUSPENDED` приведет к запуску потока в приостановленном состоянии, из которого поток может быть переведен в состояние готовности путем вызова функции `ResumeThread`.

lpThreadId — указатель на переменную типа `DWORD`, которая получает идентификатор нового потока. Если `NULL`, то идентификатор не возвращается.

Если функция выполнена успешно, то вернется описатель потока, если нет, вернется `NULL`.

Управление потоками. Примеры применения функций создания потоков

Пример 2.1. Создание нескольких потоков

Задание. Программа должна создавать два потока, выводящих в бесконечном цикле «1» и «2» соответственно. После создания дополнительных потоков, поток-родитель завершается.

В данной программе после создания потоков, главный поток завершается, способов завершения может быть два: с помощью *return* и с помощью вызова функции *ExitThread*.

Листинг 4. Файл `task_3.cpp`

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

DWORD WINAPI threadHandler(LPVOID);
int main(int argc, char* argv[])
{
    printf("Program started\n");

    HANDLE t;
    int number = 1;
    t= CreateThread(NULL, 0, threadHandler,(LPVOID)number, 0, NULL);
    CloseHandle(t);
    number = 2;
    t= CreateThread(NULL, 0, threadHandler,(LPVOID)number, 0, NULL);
    CloseHandle(t);
    // ExitThread(0); // разкомментировать для второго варианта

    printf("Program finished\n");
    system("pause");
    return 0;           // закомментировать для второго варианта
}
DWORD WINAPI threadHandler(LPVOID param){
```

```
int number = (int)param;
for(;;) {
    Sleep(1000);
    printf("%d", number);
    fflush(stdout);
}
return 0;
}
```

Результат выполнения работы программы представлен на рис. 10 и рис. 11



Рис. 10. Результат выполнения программы task_3 посредством функции ExitThread()

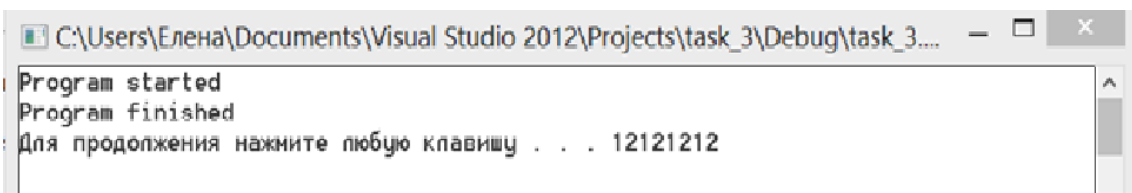


Рис. 11. Результат выполнения программы task_3.cpp при помощи return

Очевидно, что в первом случае произошло завершение всего процесса (возврат из функции main), во втором – завершился только главный поток, а процесс – нет.

Аналогичные результаты будут, если основной поток после создания потомков выполнит функцию *Sleep()* с неопределенным временем ожидания.

Функция *Sleep()* позволяет потоку отказаться от использования процессора и перейти из состояния выполнения в состояние ожидания, которое будет длиться в течение заданного промежутка времени. Например, выполнение задачи потоком может продолжаться в течение некоторого периода времени, после чего поток приостанавливается. По истечении периода ожидания планировщик вновь переводит поток в состояние готовности.

```
VOID Sleep(DWORD dwMilliseconds)
```

Длительность интервала ожидания указывается в миллисекундах, и одним из ее возможных значений является INFINITE, что соответствует бесконечному периоду ожидания, при котором выполнение приостанавливается на неопределенное время. Значению 0 соответствует отказ потока от оставшейся части отведенного ей временного промежутка (если готовых потоков нет, то будет вызван этот же поток).

Использование *Sleep()* для синхронизации является очень неудачным решением, т.к. данная функция говорит только то, что по завершению таймаута

поток/процесс перейдет в состояние «готов к выполнению». Порядок передачи управления определяет планировщик, поэтому предсказать очередность выполнения потоков не представляется возможным.

Пример 2.2. Создание программы, в которой время жизни процесса и порождаемых в нем потоков задается как параметр.

Задание. Программа должна получать 2 параметра – количество создаваемых потоков и время жизни всего приложения. С интервалом в 1 сек каждый рабочий поток выводит о себе информацию и отслеживает состояние переменной, которая устанавливается в заданное значение по истечении времени жизни процесса.

Возможны различные **варианты решения** данной задачи. Например, для подсчета времени можно использовать поток-координатор, вычисляющий момент завершения периода жизни с помощью функции *getTickCount()*, (сравнивая разницу текущего и стартового времени с заданным периодом жизни) или с помощью функции получения системного времени *GetSystemTime(&now)*. Другой способ (более рациональный) – использование таймера ожидания (Waitable Timer).

Рассмотрим вариант на основе таймера ожидания.

Таймеры ожидания(waitable timers) – это объекты ядра, которые самостоятельно переходят в свободное состояние в определенное время или через регулярные промежутки времени. Чтобы создать ожидаемый таймер, достаточно вызвать функцию *CreateWaitableTimer()*.

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

PSECURITY_ATTRIBUTES psa – атрибуты безопасности (аналогичны как и для вызовов *createThread*, *createProcess*);

BOOL fManualReset – определяет тип ожидаемого таймера (со сбросом вручную или автосбросом). Когда освобождается таймер со сбросом вручную, то возобновляется выполнение всех потоков, ожидавших этот объект, а когда в свободное состояние переходит таймер с автосбросом – лишь один из потоков.

PCTSTR pszName – имя таймера, по которому можно получить его описатель (может быть равно NULL).

Объекты «ожидаемый таймер» всегда создаются в занятом состоянии. Чтобы сообщить таймеру, в какой момент он должен перейти в свободное состояние, необходимо вызвать функцию **SetWaitableTimer**.

```
BOOL SetWaitableTimer(  
    HANDLE hTimer,  
    const LARGE_INTEGER *pDueTime,  
    LONG lPeriod,  
    PTIMERAPCROUTINE pfnCompletionRoutine,  
    PVOID pvArgToCotnpletionRoutine,  
    BOOL fResume)
```

HANDLE hTimer – описатель таймера;

const LARGE_INTEGER *pDueTime – время первого срабатывания таймера (используется специальная структура FILETIME);

LONG lPeriod – период повторений срабатывания таймера (если равен 0, то сработает 1 раз; считается в миллисекундах);

BOOL fResume – позволяет вывести компьютер из режима сна, когда таймер срабатывает (иначе – таймер перейдет в свободное состояние, но ожидавшие его потоки не получают процессорное время, пока компьютер не выйдет из режима сна).

Остальные параметры берутся по-умолчанию.

Поток – координатор получает в качестве аргумента созданный таймер и запускает функцию **WaitForSingleObject**, которая возвращает управление, если объект освобожден.

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

HANDLE hObject – описатель объекта;

DWORD dwMilliseconds – сколько времени в миллисекундах мы готовы ждать освобождения (если используется константа INFINITE, то ожидать будем бесконечно).

Чтобы посмотреть, что возвращает функция, можно использовать **GetLastError**:

WAIT_OBJECT_0 – если послано уведомление;
или WAIT_TIMEOUT – истек тайм-аут.

Листинг 5. Файл task_4.cpp

```
#include <stdio.h>  
#include <windows.h>  
  
DWORD WINAPI Thread1(LPVOID);  
  
int stop;  
struct params {
```

```

    int num;
    bool* runflg;
};

int main(int argc, char* argv[])
{
    SYSTEMTIME now;
    int thrds;
    if (argc < 3) thrds = 2;
    else thrds = atoi(argv[1]);
    if (argc < 3) stop = 5000;
    else stop = atoi(argv[2]);
    DWORD targetThreadId;
    bool runFlag = true;
    __int64 end_time;

    LARGE_INTEGER end_time2;
    //создание и установка таймера
    HANDLE tm1 = CreateWaitableTimer(NULL, false, NULL);
    end_time = -1 * stop * 10000000;
    end_time2.LowPart = (DWORD) (end_time & 0xFFFFFFFF);
    end_time2.HighPart = (LONG) (end_time >> 32);
    SetWaitableTimer(tm1, &end_time2, 0, NULL, NULL, false);

    for (int i = 0; i < thrds; i++)
    {
        params* param = (params*)malloc(sizeof(params));
        param->num = i;
        param->runflg = &runFlag;
        HANDLE t1 = CreateThread(NULL, 0, Thread1, param, 0, &targetThreadId);
        CloseHandle(t1);
    }
    GetSystemTime(&now);
    printf("System Time %d %d %d\n", now.wHour, now.wMinute, now.wSecond);
    WaitForSingleObject(tm1, INFINITE);
    runFlag = false; //установка флага
    CloseHandle(tm1);
    GetSystemTime(&now);
    printf("System Time %d %d %d\n", now.wHour, now.wMinute, now.wSecond);
    system("pause");
    return 0;
}

DWORD WINAPI Thread1(LPVOID prm) {
    while(1) {
        params arg = *((params*)prm);
        Sleep(1000);
        printf("%d\n", arg.num);
        if(*(arg.runflg) == false) //проверка флага
            break;
    }
    return 0;
}

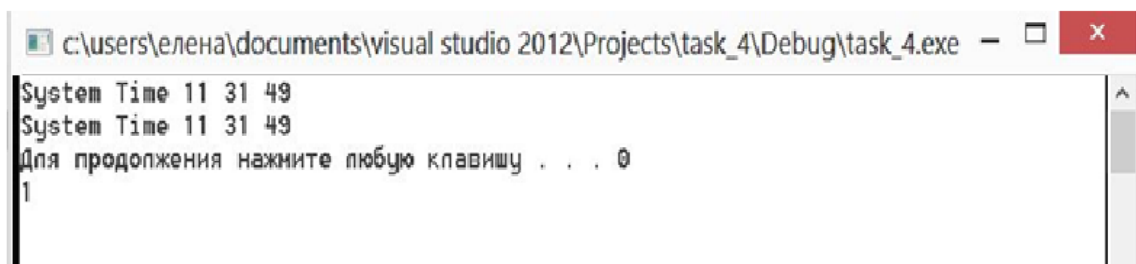
```

```
}
```

В этой программе базовый поток ожидает сигнала от таймера (по истечению заданного времени), и после этого устанавливает флаг `runFlag`, который анализируют другие потоки, и если его значение равно `false`, то они заканчивают свое выполнение.

Для контроля выполнения в начале и в конце программы выводится системное время.

Результат выполнения работы программы представлен на рис.12



```
c:\users\елена\documents\visual studio 2012\Projects\task_4\Debug\task_4.exe - [X]
System Time 11 31 49
System Time 11 31 49
Для продолжения нажмите любую клавишу . . .
1
```

Рис. 12. Результат выполнения программы `task_4.cpp`

Использовать таймер с функциями `WaitFor..Object` можно не только в разных потоках, но и в разных процессах, т.е. **ожидаемые таймеры могут служить средством межпроцессного взаимодействия.**

3. Функции управления приоритетами процессов и потоков

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 - наивысший возможный приоритет. Потоки в этом процессе обязаны немедленно реагировать на события, их исполнение может привести к полной блокировке системы и требует осторожности в использовании этого класса.

High – тоже потоки быстрого реагирования на события (этот класс присвоен Task Manager).

Above normal – класс приоритета промежуточный между `normal` и `high`, класс введенный в версии 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`, 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 потока**: idle (простаивающий), lowest (низший), below normal (ниже обычного), normal (обычный), above normal (выше обычного), highest (высший) и time-critical (критичный по времени), описания которых указаны в таблице 1.

Таблица 1. Относительные приоритеты потоков

Относительный приоритет потока	Описание
Time-critical	Поток выполняется с приоритетом 31 в классе real-time и с приоритетом 15 в других классах
Highest	Поток выполняется с приоритетом на два уровня выше обычного для данного класса
Above normal	Поток выполняется с приоритетом на один уровень выше обычного для данного класса
Normal	Поток выполняется с обычным приоритетом процесса для данного класса
Below normal	Поток выполняется с приоритетом на один уровень ниже обычного для данного класса
Lowest	Поток выполняется с приоритетом на два уровня ниже обычного для данного класса
Idle	Поток выполняется с приоритетом 16 в классе real-time и с приоритетом 1 в других классах

Относительный приоритет потока принимает значение от 0 (самый низкий) до 31 (самый высокий), но программист работает не с численными значениями, а с так называемыми «константными». Это обеспечивает определенную гибкость и независимость при изменении алгоритмов планирования, а они меняются практически с каждой новой версией ОС, а с ними, соответственно, могут измениться и соотношения приоритетов.

Таблица 2. Примерная зависимость уровня приоритета потока

Относительный приоритет потока	Класс приоритета процесса					
	Idle	Below normal	Normal	Above normal	High	Real-time
Time-critical (критичный по времени)	15	15	15	15	15	31
Highest (высший)	6	8	10	12	15	26
Above normal (выше обычного)	5	7	9	11	15	26
Normal (обычный)	4	6	8	10	13	24
Below normal (ниже обычного)	3	5	7	9	12	23
Lowest (низший)	2	4	6	8	11	22
Idle (простаивающий)	1	1	1	1	1	16

Уровень приоритета формируется самой системой, исходя из класса приоритета процесса и относительного приоритета потока

Примерный вид зависимости уровня приоритета потока от класса приоритета процесса и от относительного приоритета потока представлен в таблице 2 (на примере Windows 2000).

Динамическое повышение приоритета предназначено для оптимизации общей пропускной способности и реактивности системы, при этом выигрывает не каждое приложение в отдельности, а система в целом.

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

1. после завершения операции ввода/вывода;
2. по окончании ожидания на каком-либо объекте исполнительной системы;
3. при нехватке процессорного времени и инверсии приоритетов.

Рассмотрим более подробно каждый из этих случаев.

1) После завершения операции ввода/вывода ОС **временно** динамически повышает приоритет потоков, предоставляя им больше шансов возобновить выполнение и обработать полученные данные. После динамического повышения приоритета поток в течение одного кванта выполняется с этим приоритетом. Следующий квант потоку выделяется с понижением приоритета на один уровень. Этот цикл продолжается до тех пор, пока приоритет не снизится до базового.

2) По окончании ожидания на каком-либо объекте исполнительной системы (например, *SetEvent*, *ReleaseSemaphore*) приоритет потока увеличивается на один уровень.

3) при инверсии приоритетов диспетчер настройки баланса просматривает очереди готовых потоков и ищет потоки, которые находились в состоянии готовности (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.

Система повышает приоритет только тех потоков, **базовый** приоритет которых попадает в область **динамического** приоритета (dynamic priority range), т.е. находится в пределах 1-15. ОС не допускает динамического повышения приоритета прикладного потока до уровней реального времени (выше 15).

Системные функции обслуживаются с приоритетами реального времени.

ОС никогда не меняет приоритет потоков с уровнями реального времени (от 16 до 31). Это ограничение позволяет сохранять целостность системы и обеспечивает необходимый уровень безопасности.

Для работы с приоритетами используются следующие функции.

Функция *SetThreadPriority()* дает возможность установки базового уровня приоритета потока относительно класса приоритета его процесса.

```
BOOL SetThreadPriority(  
    HANDLE hThread, // дескриптор потока  
    int nPriority // уровень приоритета потока  
    );
```

Функция **GetThreadPriorityBoost** извлекает значение форсированного (динамически изменяемого) приоритета, который управляет состоянием заданного потока.

```
BOOL GetThreadPriorityBoost (  
    HANDLE hThread, // дескриптор потока  
    PBOOL pDisablePriorityBoost // состояние динамического изменения приоритета  
    );
```

Функция *SetThreadPriorityBoost()* разрешает/запрещает динамическое изменение приоритетов отдельного потока, не затрагивая остальные потоки.

```
BOOL SetThreadPriorityBoost (  
    HANDLE hThread, // дескриптор потока  
    BOOL DisablePriorityBoost // состояние форсирования приоритета  
    );
```

Когда поток запускается в одном из классов динамического приоритета, система временно повышает (форсирует) приоритет потока, чтобы вывести его из состояния ожидания. Если вызывается функция *SetThreadPriorityBoost()* с параметром *DisablePriorityBoost=TRUE*, приоритет потока не поднимается.

Программная установка флага **priorityBoost** позволяет для каждого из потоков отдельно указать возможность динамического повышения его приоритета. По-умолчанию, *boost* потока и процесса «разрешено».

Рабочие потоки так же могут следить за своим приоритетом, при необходимости обновляя глобальную переменную **priorityChange**.

По умолчанию ОС разрешает динамическое изменение приоритетов для всего процесса (то есть для всех его потоков). Функция *SetProcessPriorityBoost()* оказывает влияние на все потоки указанного процесса, но не препятствует дальнейшему разрешению/запрещению динамического изменения приоритетов отдельных потоков.

Иногда для передачи управления другому потоку используют команду *sleep(0)*, напомним, в этом случае нужно учитывать, что прогнозировать, какой из потоков запустится, довольно сложно, это осуществляется на усмотрение планировщика из очереди готовых в соответствии со сложившейся ситуацией в системе на текущий момент.

Пример 3. Программы для анализа влияния изменения классов приоритета процесса и принадлежащих ему потоков на выделение процессорного времени.

Задание 3.1 подготовить программу, в которой у каждого из потоков свой приоритет отличный от других. Все они выполняют одинаковую работу, например, увеличивают каждый свой счетчик. Накопленное значение счетчика, таким образом, отражает относительное суммарное время выполнения потока.

Предполагаем, так как приоритеты различны, то и время, отведенное на работу потокам различно (квант времени, выделяемый потокам, одинаков).

Структура программы см. ниже на рис.13

Листинг 7. Файл task_5.cpp

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

DWORD WINAPI Thread1(LPVOID);
int stop;
int sleep = 10000;
struct params {
    int num;
    bool* runflg;
};

long long counters[7] = {0,0,0,0,0,0,0}; //счетчики для потоков
int priority[7] = {THREAD_PRIORITY_IDLE,THREAD_PRIORITY_LOWEST,
THREAD_PRIORITY_BELOW_NORMAL,THREAD_PRIORITY_NORMAL,
THREAD_PRIORITY_ABOVE_NORMAL, THREAD_PRIORITY_HIGHEST,
THREAD_PRIORITY_TIME_CRITICAL}; //массив приоритетов в порядке
возрастания

int main(int argc, char* argv[])
{
    //в командной строке аргументом задаем время жизни потоков
    if (argc < 2) stop = 5000;
    else stop = atoi(argv[2]);

    DWORD targetThreadId;
    bool runFlag = true; //инициализация структур потока-таймера
    __int64 end_time;
    LARGE_INTEGER end_time2;
```

```

HANDLE tm1 = CreateWaitableTimer(NULL,false,NULL); //создание таймера
    end_time = -1 * stop * 10000000;
    end_time2.LowPart = (DWORD) (end_time & 0xFFFFFFFF);
    end_time2.HighPart = (LONG) (end_time >> 32);
//запуск таймера
    SetWaitableTimer(tm1, &end_time2, 0,NULL, NULL, false);
        for (int i = 0; i < 7; i++) {
            params* param = (params*)malloc(sizeof(params));
            param->num = i;
            param->runflg = &runFlag;
//порождение потока и задание ему приоритета
HANDLE t1 = CreateThread(NULL, 0, Thread1, param,0, &targetThreadId);
            SetThreadPriority(t1, priority[i]);
            PBOOL ptr1 = (PBOOL)malloc(sizeof(BOOL));
            GetThreadPriorityBoost(t1, ptr1);
//проверка динамического распределения приоритетов
            SetThreadPriorityBoost(t1, true);
            CloseHandle(t1); //очистка памяти
        }
        WaitForSingleObject(tm1,INFINITE); //ожидание потока таймера
        runFlag = false; //флаг завершения работы
        CloseHandle(tm1);
        printf("\n");
        for (int i = 0; i < 7; i++) {
            printf("%d - %ld\n",i, counters[i]); //вывод результатов
        }
        system("pause");
        return 0;
}
DWORD WINAPI Thread1(LPVOID prm)
{
    while(1) { //значение идентификатора вызывающего потока
        DWORD WINAPI thrdid = GetCurrentThreadId();
        //дескриптор потока
HANDLE WINAPI handle =
OpenThread(THREAD_QUERY_INFORMATION,false,thrdid);
        //приоритет для определяемого потока
        int WINAPI prio = GetThreadPriority(handle);
        params arg = *((params*)prm);
        counters[arg.num]++;
        if(prio != priority[arg.num])
printf("\nPriority of %d is %d %d changed\n",
        arg.num, priority[arg.num], prio); //выводится когда
//динамическое распределение проиритетов включено
        Sleep(0);
        if(*(arg.runflg) == false)
            break;
    }
    return 1;
}

```

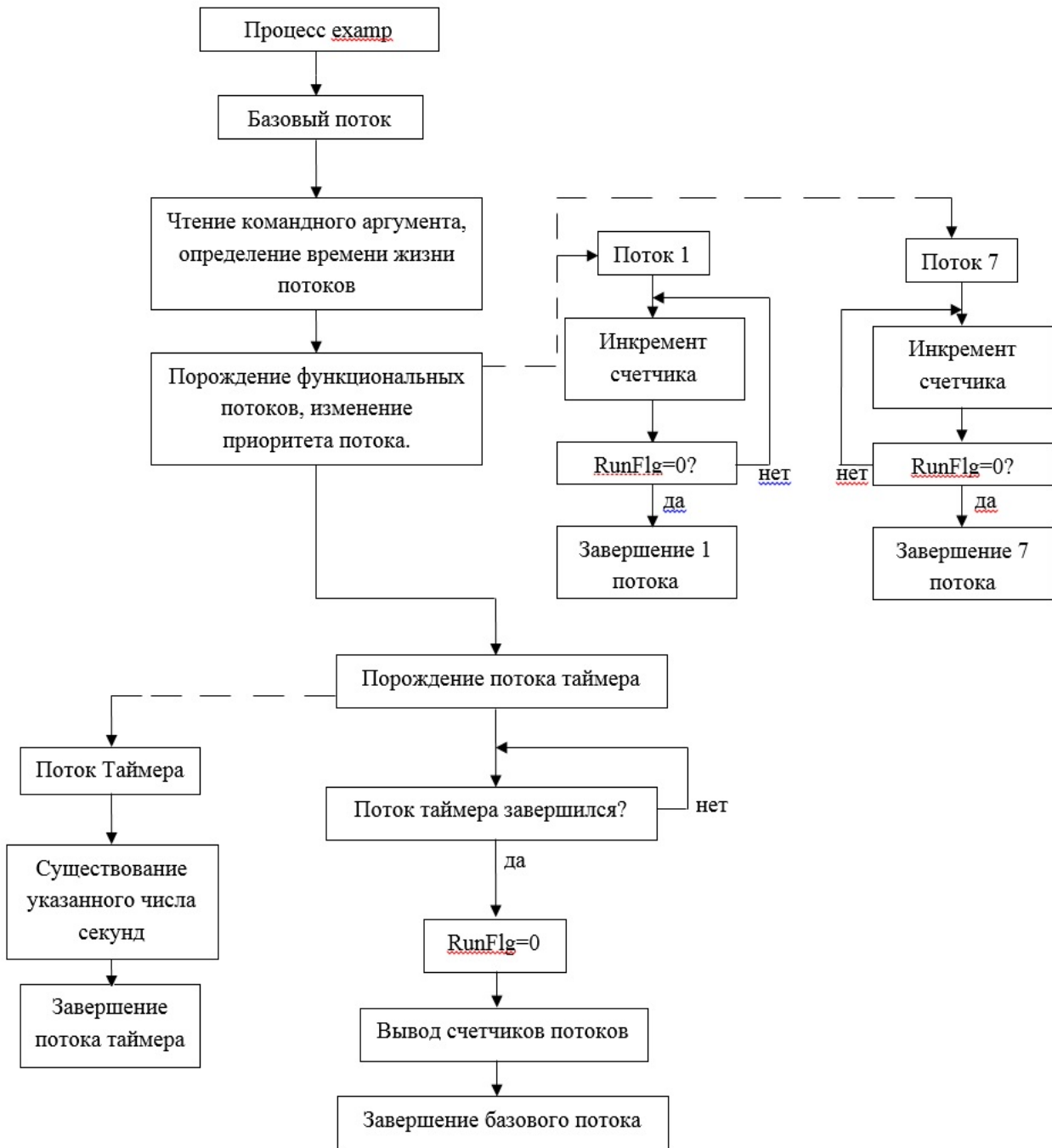



Рис. 13. Схема выполнения программы (Структура программы)

Результат выполнения работы программы представлен на рис.14



```
c:\users\елена\documents\visual studio 2012\Projects\task_5\Debug\task_5.exe
0 - 14
1 - 0
2 - 0
3 - 0
4 - 16
5 - 2
6 - 19
Для продолжения нажмите любую клавишу . . .
```

Рис. 14. Результат выполнения программы *task_5.cpp*

Слева – номер класса приоритета от низшего к высшему, а справа – значение счетчика, накопленное за все кванты, предоставленные потоку до истечения таймера. Очевидно, что потокам, у которых приоритет ниже, выделяется меньшее количество квантов времени для выполнения, и поэтому их счетчики соответственно меньше. В данном случае динамическое изменение приоритета не запрещено. При каждом запуске экспериментальные данные будут получаться различными, но пропорциональное соотношение между счетчиками потоков примерно сохраняется. Таким образом, пример демонстрирует, как ОС распределяет время процессора между потоками в зависимости от их приоритетов.

Задание 3.2. Усложним задачу и дополним ее возможностью управлять классом приоритетов процесса.

Код программы несколько изменим для получения более наглядного вывода результатов. Пусть программа по-прежнему создает 7 дополнительных потоков, со всеми возможными вариантами приоритета. Теперь в начале работы можно изменить класс приоритета процесса в целом. Каждый рабочий процесс выполняет увеличение связанного с ним счетчика (здесь типа *int*). После увеличения счетчика, поток отдает оставшуюся часть кванта времени остальным, с помощью вызова функции *Sleep* с параметром 0. Через заданное время рабочие потоки завершаются, а основной поток выводит результаты их работы в новом формате. Окончание работы происходит по сигналу от таймера. Заложена возможность задания произвольного количества потоков и времени жизни, отсчитываемого таймером.

Листинг 8. Файл *task_6.cpp*

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

#define DEF_THREADS 7
#define DEF_TTL 10
```

```

DWORD WINAPI threadHandler(LPVOID);
HANDLE initTimer(int sec);
int getPriorityIndex(DWORD prClass);

int isFinish = 0;
long counters[7] = {0,0,0,0,0,0,0};

int priorities[7] = {THREAD_PRIORITY_IDLE,THREAD_PRIORITY_LOWEST,
THREAD_PRIORITY_BELOW_NORMAL,THREAD_PRIORITY_NORMAL,
    THREAD_PRIORITY_ABOVE_NORMAL, THREAD_PRIORITY_HIGHEST,
THREAD_PRIORITY_TIME_CRITICAL};

char charPrio[7][10] = {"IDLE", "LOWEST", "BELOW", "NORMAL", "ABOVE",
"HIGHEST", "TIME_CRIT"};

char charProcPrio[6][10] = {"IDLE", "BELOW", "NORMAL", "ABOVE", "HIGH",
"REAL-TIME"};
int procPriorities[6] =
{IDLE_PRIORITY_CLASS,BELOW_NORMAL_PRIORITY_CLASS,NORMAL_PRIORITY_CLASS,
ABOVE_NORMAL_PRIORITY_CLASS,HIGH_PRIORITY_CLASS,REALTIME_PRIORITY_CLASS};

int priorityBoost[7] = {0,0,0,0,0,0,0};

int priorityChange[7] = {0,0,0,0,0,0,0};

int main(int argc, char* argv[])
{
    int numThreads = DEF_THREADS;
    int threadLive = DEF_TTL;
    if(argc < 2)
        printf("Using default numThreads = %d and default time to live
= %d\n",numThreads,threadLive);
    else if(argc < 3)
        printf("Using default time to live = %d\n",threadLive);
    else {
        numThreads = atoi(argv[1]);
        threadLive = atoi(argv[2]);
        if(numThreads <= 0 || threadLive <= 0) {
            printf("All arguments must be numbers!!!!\n");
            exit(0);
        }
    }

    HANDLE t = initTimer(threadLive);
    HANDLE t1;
    // установить приоритет процесса IDLE(или иной)
SetPriorityClass(GetCurrentProcess(),IDLE_PRIORITY_CLASS);
for (int i = 0; i < numThreads; i++) {
    t1 = CreateThread(NULL, 0, threadHandler, (LPVOID)i, 0, NULL);
    SetThreadPriority(t1, priorities[i]);
}
}

```

```

        SetThreadPriorityBoost(t1,true);
        GetThreadPriorityBoost(t1,&priorityBoost[i]);
        CloseHandle(t1);
    }

    //WaitForSingleObject(t,INFINITE); // ОЖИДАТЬ ВСЕХ ПОТОКОВ -
    КОММЕНТИРУЕМ-НЕ КОММЕНТИРУЕМ СТРОКУ В ЗАВИСИМОСТИ ОТ ПОСТАВЛЕННОЙ ЗАДАЧИ

    CloseHandle(t);
    isFinish = 1;
    char hasBoost[4];
    char wasChanged[4];
    int priorIdx =
getPriorityIndex(GetPriorityClass(GetCurrentProcess()));
    printf("Result of work:\n");
    printf("Process priority:%s\n",charProcPrio[priorIdx]);
    printf("Priority\tHas Boost\tWas changed\tCounter\n");
    for (int i = 0; i < 7; i++) {
        priorityBoost[i] == 0 ? strcpy_s(hasBoost,"NO") :
strcpy_s(hasBoost,"YES");
        priorityChange[i] == 0 ? strcpy_s(wasChanged,"NO") :
strcpy_s(wasChanged,"YES");

        printf("%8s\t%9s\t%10s\t%7d\n",charPrio[i],hasBoost,wasChanged,coun
ters[i]);
    }
    system("pause");
    return 0;
}
DWORD WINAPI threadHandler(LPVOID prm) {
    int myNum = (int)prm;
    int priority = 0;
    for(;;) {
        ++counters[myNum];
        priority = GetThreadPriority(GetCurrentThread());
        if(priority != priorities[myNum])
            priorityChange[myNum] = 1;
        if(isFinish)
            break;

            Sleep(0);
    }
    return 0;
}

HANDLE initTimer(int sec) {
    __int64 end_time;
    LARGE_INTEGER end_time2;
    HANDLE tm = CreateWaitableTimer(NULL, false, "timer");
    end_time = -1 * sec * 10000000;
    end_time2.LowPart = (DWORD) (end_time & 0xFFFFFFFF);
    end_time2.HighPart = (LONG) (end_time >> 32);
}

```

```

        SetWaitableTimer(tm, &end_time2, 0, NULL, NULL, false);
        return tm;
    }
    int getPriorityIndex(DWORD prClass) {
        for(int i = 0; i < 6; ++ i) {
            if(procPriorities[i] == prClass)
                return i;
        }
        return 0;
    }
}

```

Результаты выполнения работы программы представлены на рис.15 и рис. 16

Результат работы программы при использовании нескольких процессоров показан на рис. 15. В этом случае почти все потоки (кроме потоков с самым низким приоритетом) получили достаточное количество процессорного времени.

Результат работы программы при использовании одного процессора показан на рис. 16, потоки с самым высоким приоритетом «отбирают» процессорное время у потоков с более низким, т.е. наблюдается «ресурсное голодание».

```

Using default numThreads = 7 and default time to live = 10
Result of work:
Process priority: IDLE
Priority      Has Boost      Was changed      Counter
  IDLE        YES            NO                2790970
  LOWEST      YES            NO                3473731
  BELOW       YES            NO                4585046
  NORMAL      YES            NO                5235753
  ABOVE       YES            NO                7377776
  HIGHEST     YES            NO                10435907
  TIME_CRIT   YES            NO                11509389
Для продолжения нажмите любую клавишу . . .

```

Рис. 15. Результат выполнения task_6.cpp при использовании нескольких процессоров

```

Using default numThreads = 7 and default time to live = 10
Result of work:
Process priority: IDLE
Priority      Has Boost      Was changed      Counter
  IDLE        YES            NO                74
  LOWEST      YES            NO                377
  BELOW       YES            NO                1
  NORMAL      YES            NO                1
  ABOVE       YES            NO                91
  HIGHEST     YES            NO                16
  TIME_CRIT   YES            NO                1
Для продолжения нажмите любую клавишу . . .

```

Рис. 16. Результат выполнения программы task_6.cpp на однопроцессорной системе

В ходе эксперимента у всех потоков была включена возможность динамического изменения приоритетов операционной системой (но фактически она не использовалась).

Пример 3.3.

Задача. Анализ поведения системных функций динамического управления приоритетами процессов и потоков.

С помощью программы определим, назначается ли динамическое изменение приоритетов по умолчанию, на все ли потоки воздействует функция *SetProcessPriorityBoost()*, возможно ли разрешение отдельному потоку в процессе динамически изменять приоритет, если для процесса это запрещено.

Листинг 9. Файл task_7.cpp

```
#include <stdio.h>
#include <iostream>
#include <string>
#include <windows.h>
using namespace std;

void thread() {
    while(true) {
        Sleep(2000);
    }
}

int main(int argc, char *argv[]) {

    BOOL dynamic;
    HANDLE processHandle, mainThread, secondThread;
    DWORD secondID;
    processHandle = GetCurrentProcess();
    mainThread = GetCurrentThread();

    //create a second thread
    secondThread = CreateThread(NULL,0, (LPTHREAD_START_ROUTINE)
thread,NULL,NULL,&secondID);

    //print default values
    cout <<"0 - dynamic enable, 1-disabled." <<endl;
    GetProcessPriorityBoost(processHandle,&dynamic);
    cout <<"Process dynamically default is " <<dynamic <<endl;
    GetThreadPriorityBoost(mainThread,&dynamic);
    cout <<"Main thread dynamic default is " <<dynamic <<endl;
    GetThreadPriorityBoost(secondThread,&dynamic);
    cout <<"Second thread dynamic default is " <<dynamic <<endl
<<endl;

    cout <<"Change Second thread priority" <<endl;
    if(!SetThreadPriorityBoost(secondThread,true)){ //dizable
        cout <<"Error on thread change!" <<endl;
        return 2;
    }
}
```

```

}

GetProcessPriorityBoost(processHandle,&dynamic);
cout <<"Process dynamically default is " <<dynamic <<endl;
GetThreadPriorityBoost(mainThread,&dynamic);
cout <<"Main thread dynamic default is " <<dynamic <<endl;
GetThreadPriorityBoost(secondThread,&dynamic);
cout <<"Second thread dynamic default is " <<dynamic <<endl
<<endl;

//change process dinamically
if(!SetProcessPriorityBoost(processHandle,true)) {
    cout <<"Error on prir change!" <<endl;
    return 1;
}

cout <<"After process dynamic drop:" <<endl;
GetProcessPriorityBoost(processHandle,&dynamic);
cout <<"Process is " <<dynamic <<endl;
GetThreadPriorityBoost(mainThread,&dynamic);
cout <<"Main thread is " <<dynamic <<endl;
GetThreadPriorityBoost(secondThread,&dynamic);
cout <<"Second thread is " <<dynamic <<endl <<endl;

//may be can change thread dynamic prio?
if(!SetThreadPriorityBoost(secondThread,false)){ //dizable
    cout <<"We cannot change if process ban dynamic prio!!!"
<<endl;
    return 3;
} else {
    cout <<"Thread dynamic prio changed, but process bans
it!!!" <<endl;
    GetProcessPriorityBoost(processHandle,&dynamic);
    cout <<"Process is " <<dynamic <<endl;
    GetThreadPriorityBoost(mainThread,&dynamic);
    cout <<"Main thread is " <<dynamic <<endl;
    GetThreadPriorityBoost(secondThread,&dynamic);
    cout <<"Second thread is " <<dynamic <<endl;
}
system("pause");
return 0;
}

```

Результат выполнения работы программы представлен на рис.17

```

c:\users\елена\documents\visual studio 2012\Projects\task_7\Debug\task_7.exe
0 - dynamic enable, 1-disabled.
Process dynamically default is 0
Main thread dynamic default is 0
Second thread dynamic default is 0

Change Second thread priority
Process dynamically default is 0
Main thread dynamic default is 0
Second thread dynamic default is 1

After process dynamic drop:
Process is 1
Main thread is 1
Second thread is 1

Thread dynamic prio changed, but process bans it!!!
Process is 1
Main thread is 1
Second thread is 0
Для продолжения нажмите любую клавишу . . .

```

Рис. 17. Результат выполнения программы task_7.cpp

Выполнить самостоятельно:

1. Исследуйте результаты работы программы 3.1 и 3.2 в зависимости от того, какой приоритет назначается базовому потоку: `above_normal`, `idle_priority class`, `high priority class`, `normal priority class` и др.; своими экспериментальными данными заполните таблицу 3, приведенную ниже, с точным указанием для какой ОС и на каком отладочном комплексе проводились измерения;

Таблица 3. Пример зависимости предоставляемого процессорного ресурса от уровня приоритета потока, задаваемого двумя параметрами

Относительный приоритет потока	Класс приоритета процесса					
	Idle	Below normal 16	Normal 32	Above normal	High 128	Real-time
THREAD_PRIORITY_IDLE,		13	3404	0	8953	
THREAD_PRIORITY_LOWEST,		63905	17071	0	91836	
THREAD_PRIORITY_BELOW_NORMAL,		44386	89864	54928	48390	
THREAD_PRIORITY_NORMAL,		265477	296141	337473	279060	
THREAD_PRIORITY_ABOVE_NORMAL,		390263	352937	396033	392205	
THREAD_PRIORITY_HIGHEST,		373955	380672	396811	362524	
THREAD_PRIORITY_TIME_CRITICAL		392055	372059	400663	392844	

2. Модифицируйте программу 3.2 для заполнения таблицы 2 текущими данными вашего эксперимента. Сделайте выводы;
3. С помощью утилит CPU Stress, позволяющих нагружать систему, и утилиты мониторинга ProcessExplorer() (или иных утилит) зафиксируйте динамическое изменение приоритетов, приведите результаты в отчете;
4. Создайте программу, демонстрирующую возможность **наследования**

- a) дескриптора порождающего процесса,
- b) дескрипторов открытых файлов,

для выполнения этого задания следует учесть, что по умолчанию наследование в Windows отключено и для возможности наследования, необходимо:

- 1) разрешить процессу-потомку наследовать дескрипторы,
- 2) сделать дескрипторы наследуемыми

(более подробно см. раздел 4, параграф 1, стр.65).

Раздел 3. Средства межпроцессного взаимодействия в ОС Windows

В этом разделе мы изучим такие средства взаимодействия потоков и процессов (IPC) в ОС Windows как: именованные и неименованные каналы; разделяемая память и семафоры; сигналы; сокет и др.

Для упрощения восприятия и сравнения перечисленных средств рассмотрим их применение на примере клиент-серверного приложения (для каждого типа IPC), в котором (по возможности) сервер должен передавать сообщение клиента обратно клиенту (эхо-сервер).

1. Неименованные каналы (Pipe)

Посредством pipe-канала можно передавать данные только между двумя процессами. В основе взаимодействия лежит так называемая **файловая модель функционирования**. Один из процессов создает канал, другой открывает его. После этого оба процесса могут передавать данные через канал в одну или обе стороны, используя для этого функции, предназначенные для работы с файлами, такие как ReadFile и WriteFile.

Анонимные каналы (anonymous channels) Windows обеспечивают однонаправленное (полудуплексное) посимвольное межпроцессное взаимодействие. Каждый канал имеет два дескриптора: дескриптор чтения (read handle) и дескриптор записи (write handle).

Функция, с помощью которой **создаются анонимные каналы**, имеет следующий прототип:

```
BOOL CreatePipe(PHANDLE phRead, PHANDLE phWrite,  
                LPSECURITY_ATTRIBUTES lpsa, DWORD nsize)
```

PHANDLE phRead, PHANDLE phWrite – указатели на дескрипторы чтения и записи соответственно;

LPSECURITY_ATTRIBUTES lpsa – атрибуты защиты (там указывается и флаг разрешения наследования дескриптора);

DWORD nsize – размер канала в байтах. Носит рекомендательный характер (конкретный размер будет определен системой). При значении равном 0 используется размер по-умолчанию.

Возвращаемое значение: 0 – при ошибке, не 0 при нормальной работе.

Для получения данных из канала используется функция:

```
BOOL WINAPI ReadFile(HANDLE hFile,  
                    LPVOID lpBuffer,  
                    DWORD nNumberOfBytesToRead,  
                    LPDWORD lpNumberOfBytesRead,  
                    LPOVERLAPPED lpOverlapped);
```

HANDLE hFile – дескриптор файла, из которого производится чтение;

LPVOID lpBuffer – указатель на буфер-приемник данных;

DWORD nNumberOfBytesToRead – максимальное число читаемых байт (размер буфера);

LPDWORD lpNumberOfBytesRead – указатель на переменную, в которую заносится число фактически прочитанных байт. Этот параметр может быть равен 0, если в момент чтения указатель находился на конце файла либо если функция завершилась с ошибкой;

LPOVERLAPPED lpOverlapped – специальная структура, позволяющая организовать асинхронный ввод-вывод. Она содержит в своем составе позицию, с которой будет производиться операция в файле и событие (его дескриптор), которое будет переходить в сигнальное состояние по завершению выполнения операции (сброс ручной).

Возвращаемое значение: в случае успешного выполнения (которое считается таковым, даже если не был считан ни один байт из-за попытки чтения с выходом за пределы файла) — TRUE, иначе — FALSE.

Функция возвращает управление в следующих ситуациях: заверена операция записи на дескрипторе записи канала, прочитано требуемое количество данных или в случае возникновения ошибки.

Для записи данных в канал используется функция:

```
BOOL WriteFile(HANDLE hFile, LPCVOID lpBuffer,  
              DWORD nNumberOfBytesToWrite, LPDWORD lpNumberOfBytesWritten,  
              LPOVERLAPPED lpOverlapped)
```

Все параметры аналогичны функции `ReadFile` (только вместо чтения производится запись и указывается буфер, из которого будут забираться данные).

Возвращаемое значение: в случае успешного выполнения — `TRUE`, иначе — `FALSE`.

Если функция возвращает управление, это еще не означает, что данные передались на диск, если только файл не был создан с флагом `FILE_FLAG_WRITE_THROUGH`.

Функция не вернет управления, пока все байты не будут записаны (т.е. пока в буфере не будет достаточно для них места).

После создания канала необходимо передать клиентскому процессу его дескрипторы (или один из них), что обычно делается с помощью **механизма наследования**.

Напомним, что для **наследования описателя** нужно, чтобы процесс-потомок создавался функцией `CreateProcess` с флагом наследования `TRUE`. Предварительно нужно создать наследуемые описатели. Сделать это можно несколькими способами.

Например, путем явной спецификации параметра `bInheritHandle` структуры `SECURITY_ATTRIBUTES` при создании канала. Поскольку процессу-потомку значение наследуемого дескриптора пока еще не известно, родительский процесс должен передать это значение потомку либо через механизм межпроцессного взаимодействия (`Interprocess Communication, IPC`), либо путем назначения дескриптора стандартному устройству ввода/вывода в структуре `STARTUPINFO`. Последний вариант позволяет произвести передачу дескрипторов без внесения изменений в дочернюю программу.

Другим способом является создание наследуемого дубликата имеющегося описателя при помощи функции `DuplicateHandle` и последующая передача его создаваемому процессу через командную строку или каким-либо иным образом (через `IPC`). Один из возможных примеров реализации приведен в предыдущем параграфе.

Пример 1. Программа, демонстрирующая передачу данных неименованными каналами

Задание. Создать клиент-серверное приложение, позволяющее набираемые символы в терминальном окне командной строки (сервер) отображать их в окне процесса-потомка (клиент).

В программе-сервере *master* создается неименованный канал для связи с процессом-потомком, порождается сам процесс-потомок (программа-клиент) *slave*. На стороне сервера производится запись из консоли в канал. В *slave* открывается неименованный канал и осуществляется считывание из него в новое окно. Запись\чтение канала производится с помощью стандартных потоков `std_in` и `std_out`.

Исходный код сервера "master"

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <iostream>
using namespace std;

int main()
{    // инициализируем необходимые структуры
    STARTUPINFO si = {sizeof(si)};
    SECURITY_ATTRIBUTES sa;
    PROCESS_INFORMATION pi;
    char buf[1024];
    char t= '\n';
    HANDLE newstdread, newstdwrite;//хэндлы потоков для пайпа
    //инициализируем нужные поля SECURITY_ATTRIBUTES
    sa.nLength = sizeof(sa);
    sa.lpSecurityDescriptor = NULL;
    sa.bInheritHandle = true;        //разрешаем наследование дескрипторов
    //создаем анонимный канал(создаем пайп для stdin)
    if (!CreatePipe(&newstdread, //указатель на переменную типа dword,
//которая получит хэндл конца чтения пайпа
                    &newstdwrite, //указатель на переменную типа dword, которая
получит хэндл на конец записи пайпа
                    &sa, // указатель на структуру атрибутов безопасности
                    0)) //размер буфера, используется по умолчанию
    {
        cout << "I can't CreatePipe";
        getch();
        return 0;
    }
    else
        cout << "\nPipe Created!\n";
    //выводим на экран дескриптор потока ввода анонимного канала
    cout << "The read HANDLE of PIPE = " << newstdread << endl;
    //обнуляем поля STARTUPINFO и задаем нужные значения
    ZeroMemory(&si, sizeof(STARTUPINFO));
    si.cb = sizeof(STARTUPINFO);
    si.dwFlags = STARTF_USESTDHANDLES | STARTF_USESHOWWINDOW;
    si.wShowWindow = SW_NORMAL;
```

```

        //подменяем стандартный дескриптор ввода дескриптором ввода канала
        si.hStdInput = newstdread;
        si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);
        si.hStdError = si.hStdOutput;
        TCHAR czCommandLine[] = L"c:\\\\anonymous_pipe_client.exe";
        if (!CreateProcess(NULL, czCommandLine, NULL, NULL, TRUE,
CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi))
        {
            cout << "Error: Can't CreateProcess";
            getch();
            CloseHandle(newstdread);
            CloseHandle(newstdwrite);
            return 0;
        }
        else
        cout << "\nProcess Created!!!\n";
        memset(buf, '\\0', sizeof(buf));
        cout << buf;
        unsigned long bread;
        cout << "STD INPUT HANDLE = " <<
GetStdHandle(STD_INPUT_HANDLE) << endl;
        cout << "STD OUTPUT HANDLE = " <<
GetStdHandle(STD_OUTPUT_HANDLE) << endl;
        while(1)
        {
            memset(buf, '\\0', sizeof(buf));
            *buf = (char)getch();
            cout.put(*buf);
            if(*buf==13)
            {
                *buf = '\\n';
                cout.put(*buf);
            }
            WriteFile(newstdwrite, //указатель на пишущих хэндл канала
                buf, // указатель на буфер
                1, //кол-во байт данных,записываемых в буфер
                &bread, // указатель на переменную, хранящую кол-во байт,
//записанных в буфер
                NULL); //т.к. 1-й аргумент не был открыт с флагом
FILE_FLAG_OVERLAPPED
            if(*buf==27)
                break;
        }
        TerminateProcess(pi.hProcess, 0); // завершение процесса
        CloseHandle(pi.hThread);
        CloseHandle(pi.hProcess);
        CloseHandle(newstdread);
        CloseHandle(newstdwrite);
        system("PAUSE");
        return 0;}

```

Исходный код клиента "slave"

```
#include <iostream>
```

```

#include <conio.h>
#include <stdio.h>
#include <windows.h>
using namespace std;
int main()
{
    char buf[2];
    unsigned long avail;
    cout << "STD INPUT HANDLE = " <<
GetStdHandle(STD_INPUT_HANDLE) << "\n";
    cout << "STD OUTPUT HANDLE = " <<
GetStdHandle(STD_OUTPUT_HANDLE) << "\n";
    unsigned long bread; //кол-во прочитанных байт
    while(1){
        PeekNamedPipe( // получаем данные из анонимного канала
GetStdHandle(STD_INPUT_HANDLE), // идентификатор канала Pipe
        NULL, //адрес буфера для прочитанных данных(NULL -нет данных для чтения)
        NULL, //размер буфера в байтах, параметр игнорируется, если буфер NULL
        NULL, //указатель на переменную, которая получает число считанных байт
        //параметр = NULL, если нет данных для чтения
        &avail, //адрес переменной, в которую будет записано общее количество байт данных,
        //доступных в канале для чтения
        NULL); // адрес переменной, в которую будет записано количество непрочитанных байт
        // в данном сообщении
        if(avail)
        {
            memset(buf, '\0', sizeof(buf));
            ReadFile(
                GetStdHandle(STD_INPUT_HANDLE), //handle канала
                buf, //указатель на буфер, в который пишем
                считанные из //канала данные
                1, //кол-во байтов для чтения
                &bread, //указатель на переменную (кол-во
                считанных байт)
                NULL); // если handle не OVERLAPPED, то
                равен NULL
                cout << buf;
            }
        }
    }
    return 0;
}

```

Результаты выполнения

При запуске "сервера" в командной строке выводится текст «Pipe Created!», затем сервером порождается процесс "anonymous_pipe_client" в новом окне, после чего любые символы, которые пишем в окне сервера, моментально появляются в окне клиента.

Рис.18. Результаты работы приложения с IPC pipe и консолью

Unnamed pipe является однонаправленным локальным средством взаимодействия процессов с наследованием или родственными, как в данном случае.

Пример 2. Создать эхо-сервер, взаимодействующий с клиентом посредством pipe

Описание программы

В программе используется передача дескрипторов через наследование. По причине того, что анонимный канал является полудуплексным, для организации эхо-сервера необходимо создавать 2 канала (для передачи от клиента-серверу и обратно). При этом ненужные дескрипторы каналов закрываются только на стороне сервера (т.к. клиент наследует 4 дескриптора, а явно мы передаем только 2 дескриптора).

Дескрипторы каналов связываются со стандартным вводом и выводом клиентского процесса. Поэтому клиент выводит информацию в поток ошибок (что приведет к выводу в консоль процесса-клиента).

Клиент передает сообщение, например, вида: «message num 1». Сервер передает данное сообщение обратно. Процессы завершаются после передачи 10 сообщений.

Исходный код сервера (содержимое файла anonymousPipesMainProc.cpp):

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

int main(int argc, char* argv[])
{
    HANDLE hReadPipeFromServToClient, hWritePipeFromServToClient; //дескрипторы канала для
    //передачи от сервера клиенту
    HANDLE hReadPipeFromClientToServ, hWritePipeFromClientToServ; //дескрипторы канала для
    //передачи от сервера клиенту
    SECURITY_ATTRIBUTES PipeSA = {sizeof(SEcurity_ATTRIBUTES), NULL, TRUE}; //чтобы сделать
    //дескрипторы наследуемыми
    //создаем канал для передачи от сервера клиенту, сразу делаем дескрипторы наследуемыми
    if(CreatePipe(&hReadPipeFromServToClient,&hWritePipeFromServToClient,&PipeSA,0)==0)
    {
        printf("impossible to create anonymous pipe from serv to client\n");
    }
}
```

```

    getchar();
    return 1000;
}
//создаем канал для передачи от клиента серверу, сразу делаем дескрипторы наследуемыми
if(CreatePipe(&hReadPipeFromClientToServ,&hWritePipeFromClientToServ,&PipeSA,0)==0)
{
    printf("impossible to create anonymous pipe from client to serv\n");
    getchar();
    return 1001;
}
PROCESS_INFORMATION processInfo_Client; // информация о процессе-клиенте
STARTUPINFO startupInfo_Client; //структура, которая описывает внешний вид основного
//окна и содержит дескрипторы стандартных устройств нового процесса, используем для установки
//процесс-клиент будет иметь те же параметры запуска, что и сервер, за исключением //дескрипторов
//ввода, вывода и ошибок
GetStartupInfo(&startupInfo_Client);
startupInfo_Client.hStdInput=hReadPipeFromServToClient; //устанавливаем поток ввода
startupInfo_Client.hStdOutput=hWritePipeFromClientToServ; //установим поток вывода
startupInfo_Client.hStdError=GetStdHandle(STD_ERROR_HANDLE); //установим поток ошибок
startupInfo_Client.dwFlags = STARTF_USESTDHANDLES; //устанавливаем наследование
//создаем процесс клиента
CreateProcess(NULL, "..\\..\\anonymousPipesSonProc\\Release\\anonymousPipesSonProc.exe", NULL,
    NULL, TRUE, CREATE_NEW_CONSOLE, NULL, NULL,
    &startupInfo_Client, &processInfo_Client);
CloseHandle(processInfo_Client.hThread); //закрываем дескрипторы созданного процесса и его
//потока
CloseHandle(processInfo_Client.hProcess);
//закрываем ненужные дескрипторы каналов, которые не использует сервер
CloseHandle(hReadPipeFromServToClient);
CloseHandle(hWritePipeFromClientToServ);

#define BUF_SIZE 100 //размер буфера для сообщений
BYTE buf[BUF_SIZE]; //буфер приема/передачи
DWORD readbytes, writebytes; //число прочитанных/переданных байт
for(int i=0; i<10; i++)
{
    //читаем данные из канала от клиента
    if(!ReadFile(hReadPipeFromClientToServ, buf, BUF_SIZE, &readbytes, NULL))
    {
        printf("impossible to use readfile\n GetLastError= %d\n", GetLastError());
        getchar();
        return 1000;
    }
    printf("get from client: \"%s\"\n", buf);
    if(!WriteFile(hWritePipeFromServToClient, buf, readbytes, &writebytes, NULL))
    {
        printf("impossible to use writefile\n GetLastError= %d\n", GetLastError());
        getchar();
        return 1001;
    }
    //пишем данные в канал клиенту
}
//закрываем HANDLE каналов
CloseHandle(hReadPipeFromClientToServ);
CloseHandle(hWritePipeFromServToClient);

printf("server ended work\n Press any key");
getchar();
return 0;
}

```

Исходный код клиента (содержимое файла anonymousPipesSonProc.cpp):

```

#include <stdio.h>
#include <Windows.h>

int main(int argc, char* argv[])
{

```



```

char strtosend[100]; //строка для передачи
char getbuf[100]; //буфер приема
int bytestosend; //число передаваемых байт
DWORD bytestosend,bytesreaded; //число переданных и принятых байт
for(int i=0;i<10;i++)
{
    bytestosend=sprintf(strtosend,"message num %d",i+1); //формирование строки для
//передачи
    strtosend[bytestosend]=0;
    fprintf(stderr,"client sended: \"%s\"\n",strtosend);

    if(!WriteFile(GetStdHandle(STD_OUTPUT_HANDLE),strtosend,bytestosend+1,&bytesreaded,NULL))
//передача данных
    {
        fprintf(stderr,"Error with writeFile\n Wait 5 sec GetLastError=
%d\n",GetLastError());
        Sleep(5000);
        return 1000;
    }
    if(!ReadFile(GetStdHandle(STD_INPUT_HANDLE),getbuf,100,&bytesreaded,NULL))
//прием ответа от сервера
    {
        fprintf(stderr,"Error with readfile\n Wait 5 sec GetLastError=
%d\n",GetLastError());
        Sleep(5000);
        return 1001;
    }
    fprintf(stderr,"Get msg from server: \"%s\"\n",getbuf);
}
fprintf(stderr,"client ended work\n Wait 5 sec");
Sleep(5000);
return 0;
}

```

Результаты выполнения программы

```

get from client: "message num 1"
get from client: "message num 2"
get from client: "message num 3"
get from client: "message num 4"
get from client: "message num 5"
get from client: "message num 6"
get from client: "message num 7"
get from client: "message num 8"
get from client: "message num 9"
get from client: "message num 10"
server ended work
Press any key

client sended: "message num 1"
Get msg from server: "message num 1"
client sended: "message num 2"
Get msg from server: "message num 2"
client sended: "message num 3"
Get msg from server: "message num 3"
client sended: "message num 4"
Get msg from server: "message num 4"
client sended: "message num 5"
Get msg from server: "message num 5"
client sended: "message num 6"
Get msg from server: "message num 6"
client sended: "message num 7"
Get msg from server: "message num 7"
client sended: "message num 8"
Get msg from server: "message num 8"
client sended: "message num 9"
Get msg from server: "message num 9"
client sended: "message num 10"
Get msg from server: "message num 10"
client ended work
Wait 5 sec

```

Рис. 19. Результат работы приложения с эхо-сервером на основе pipe

2. Именованные каналы

Информация об используемом средстве IPC

Именованные каналы являются дуплексными, ориентированы на обмен сообщениями и обеспечивают взаимодействие через сеть. Кроме того, один

именованный канал может иметь несколько открытых дескрипторов. В сочетании с удобными, ориентированными на выполнение транзакций функциями эти возможности делают именованные каналы пригодными для создания клиент-серверных систем. Обмен данными может быть синхронным и асинхронным.

Для создания именованного канала используется функция:

```
HANDLE CreateNamedPipe(LPCTSTR lpName, DWORD dwOpenMode,  
DWORD dwPipeMode, DWORD nMaxInstances, DWORD nOutBufferSize,  
DWORD nInBufferSize, DWORD nDefaultTimeOut,  
LPSECURITY_ATTRIBUTES lpSecurityAttributes)
```

LPCTSTR lpName - указатель на имя канала, который должен иметь следующую форму:

«\\.\pipe\[path]pipename»

Точка (.) обозначает локальный компьютер; таким образом, создать канал на удаленном компьютере невозможно;

DWORD dwOpenMode – задает режим, в котором открывается канал. Может иметь следующие значения (которые могут комбинироваться):

- PIPE_ACCESS_INBOUND - только для чтения (передача от клиента-серверу, сервер открывает pipe с параметром чтения, клиент – записи);
- PIPE_ACCESS_OUTBOUND - только для записи;
- PIPE_ACCESS_DUPLEX - для чтения и записи;
- FILE_FLAG_OVERLAPPED - использование асинхронных операций (ввод и вывод с перекрытием). Данный режим позволяет процессу выполнять полезную работу параллельно с проведением операций в канале;
- FILE_FLAG_WRITE_THROUGH - в этом режиме функции, работающие с каналом, не возвращают управление до тех пор, пока не будет полностью завершена операция на удаленном компьютере. Используется только с каналом, ориентированным на передачу отдельных байт и только в том случае, когда канал создан между процессами, запущенными на различных станциях сети;

DWORD dwPipeMode – имеются три пары взаимоисключающих значений этого параметра. Эти значения указывают, ориентирована ли запись на работу с сообщениями или байтами, ориентировано ли чтение на работу с сообщениями или блоками, и блокируются ли операции чтения.

- PIPE_TYPE_BYTE и PIPE_TYPE_MESSAGE — указывают, соответственно, должны ли данные записываться в канал как поток байтов или как сообщения. Для всех экземпляров каналов с одинаковыми именами следует использовать одно и то же значение.
- PIPE_READMODE_BYTE и PIPE_READMODE_MESSAGE — указывают, соответственно, должны ли данные считываться как поток байтов или как сообщения. Значение PIPE_READMODE_MESSAGE требует использования значения PIPE_TYPE_MESSAGE.
- PIPE_WAIT и PIPE_NOWAIT — определяют, соответственно, будет или не будет блокироваться операция ReadFile. Следует использовать значение PIPE_WAIT, поскольку для обеспечения асинхронного ввода/вывода существуют лучшие способы.

DWORD nMaxInstances – определяет количество экземпляров каналов, а следовательно, и количество одновременно подключенных клиентов;

DWORD nOutBufferSize, DWORD nInBufferSize – позволяют указать размеры в байтах входного и выходного буферов канала. Если равны 0, то используются размер по-умолчанию;

DWORD nDefaultTimeOut – длительность интервала ожидания по-умолчанию для функции WaitNamedPipe;

LPSECURITY_ATTRIBUTES lpSecurityAttributes – атрибуты защиты.

Возвращаемое значение: в случае успеха возвращается дескриптор канала со стороны сервера, в случае неудачи - INVALID_HANDLE_VALUE.

При первом вызове функции CreateNamedPipe происходит создание самого именованного канала, а не просто его экземпляра. Закрытие последнего открытого дескриптора экземпляра именованного канала приводит к уничтожению этого экземпляра (обычно существует по одному дескриптору на каждый экземпляр). Уничтожение последнего экземпляра именованного канала приводит к уничтожению самого канала, в результате чего имя канала становится вновь доступным для повторного использования.

После создания именованного канала сервер может ожидать подключения клиента, вызывая функцию:

```
BOOL ConnectNamedPipe(HANDLE hNamedPipe, LPOVERLAPPED lpOverlapped);
```

HANDLE hNamedPipe – дескриптор именованного канала;

LPOVERLAPPED lpOverlapped – указатель на структуру OVERLAPPED.

Возвращаемое значение – в случае успеха – TRUE, в случае неудачи – FALSE.

Функция возвращает управление после соединения с клиентом (если второй параметр функции равен NULL). После сервер обменивается с клиентом данными при помощи функций WriteFile и ReadFile.

По завершению обмена сервер вызывает функцию:

```
BOOL DisconnectNamedPipe(HANDLE hNamedPipe);
```

HANDLE hNamedPipe – дескриптор именованного канала.

Возвращаемое значение – в случае успеха – TRUE, в случае неудачи – FALSE.

Для подключения клиента к именованному каналу применяется функция **CreateFile**, при вызове которой указывается имя именованного канала:

```
HANDLE CreateFile( LPCTSTR lpFileName, DWORD dwDesiredAccess, DWORD dwShareMode, LPSECURITY_ATTRIBUTES lpSecurityAttributes, DWORD dwCreationDistribution, DWORD dwFlagsAndAttributes, HANDLE hTemplateFile);
```

LPCTSTR lpFileName – имя канала. Часто клиент и сервер выполняются на одном компьютере, и в этом случае для указания имени канала используется следующая форма:

«\\.\pipe\[path]pipename»

Если сервер находится на другом компьютере, для указания имени канала используется следующая форма:

«\\servername\pipe\[path]pipename»

Использование точки (.) вместо имени локального компьютера в случае, когда сервер является локальным, позволяет значительно сократить время подключения;

DWORD dwDesiredAccess – тип доступа. Можно использовать комбинацию из следующих констант:

- 0 - доступ запрещен, однако приложение может определять атрибуты файла, канала или устройства, открываемого при помощи функции CreateFile XE "CreateFile";
- GENERIC_READ - разрешен доступ на чтение из файла или канала Pipe;
- GENERIC_WRITE - разрешен доступ на запись в файл или канал Pipe;

DWORD *dwShareMode* – с помощью данного параметра задаются режимы совместного использования открываемого или создаваемого файла. Можно указать комбинацию следующих констант:

- 0 - совместное использование файла запрещено;
- FILE_SHARE_READ - другие приложения могут открывать файл с помощью функции CreateFile для чтения;
- FILE_SHARE_WRITE - аналогично предыдущему, но на запись;

LPSECURITY_ATTRIBUTES *lpSecurityAttributes* – атрибуты защиты;

DWORD *dwCreationDistribution* - определяет действия, выполняемые функцией CreateFile, если приложение пытается создать файл, который уже существует. Могут быть указаны следующие константы:

- CREATE_NEW - если создаваемый файл уже существует, функция CreateFile возвращает код ошибки;
- CREATE_ALWAYS - существующий файл перезаписывается, при этом содержимое старого файла теряется;
- OPEN_EXISTING - открывается существующий файл. Если файл с указанным именем не существует, функция CreateFile возвращает код ошибки;
- OPEN_ALWAYS - если указанный файл существует, он открывается. Если файл не существует, он будет создан;
- TRUNCATE_EXISTING - если файл существует, он открывается, после чего длина файла устанавливается равной нулю. Содержимое старого файла теряется. Если же файл не существует, функция CreateFile возвращает код ошибки;

DWORD *dwFlagsAndAttributes* – задает атрибуты и флаги для файла. В качестве атрибутов могут быть указаны следующие:

- FILE_ATTRIBUTE_ARCHIVE - файл был архивирован (выгружен);
- FILE_ATTRIBUTE_COMPRESSED - файл, имеющий этот атрибут, динамически сжимается при записи и восстанавливается при чтении. Если этот атрибут имеет каталог, то для всех расположенных в нем файлов и каталогов также выполняется динамическое сжатие данных;
- FILE_ATTRIBUTE_NORMAL - остальные перечисленные в этом списке атрибуты не установлены;
- FILE_ATTRIBUTE_HIDDEN - скрытый файл;
- FILE_ATTRIBUTE_READONLY - файл можно только читать;
- FILE_ATTRIBUTE_SYSTEM - файл является частью операционной системы.

В качестве флагов можно указать следующие:

- `FILE_FLAG_WRITE_THROUGH` - отмена промежуточного кэширования данных для уменьшения вероятности потери данных при аварии;
- `FILE_FLAG_NO_BUFFERING` - отмена промежуточной буферизации или кэширования. При использовании этого флага необходимо выполнять чтение и запись порциями, кратными размеру сектора (обычно 512 байт);
- `FILE_FLAG_OVERLAPPED` - асинхронное выполнение чтения и записи. Во время асинхронного чтения или записи приложение может продолжать обработку данных;
- `FILE_FLAG_RANDOM_ACCESS` - указывает, что к файлу будет выполняться произвольный доступ. Флаг предназначен для оптимизации кэширования;
- `FILE_FLAG_SEQUENTIAL_SCAN` - указывает, что к файлу будет выполняться последовательный доступ от начала файла к его концу. Флаг предназначен для оптимизации кэширования;
- `FILE_FLAG_DELETE_ON_CLOSE` - файл будет удален сразу после того как приложение закроет его идентификатор. Этот флаг удобно использовать для временных файлов;
- `FILE_FLAG_BACKUP_SEMANTICS` - файл будет использован для выполнения операции выгрузки или восстановления. При этом выполняется проверка прав доступа;
- `FILE_FLAG_POSIX_SEMANTICS` - доступ к файлу будет выполняться в соответствии со спецификацией POSIX;

HANDLE hTemplateFile – предназначен для доступа к файлу шаблона с расширенными атрибутами создаваемого файла.

Возвращаемое значение: в случае успешного завершения функция возвращает идентификатор открытого файла/канала, в случае неудачи - `INVALID_HANDLE_VALUE`.

С помощью функции **WaitNamedPipe** процесс может выполнять ожидание момента, когда канал Pipe будет доступен для соединения:

```
BOOL WaitNamedPipe( LPCTSTR lpszPipeName,  DWORD  dwTimeout);
```

LPCTSTR lpszPipeName – имя канала;

DWORD dwTimeout – время ожидания (в миллисекундах). В качестве параметра могут быть переданы константы: `NMPWAIT_WAIT_FOREVER` -

ожидание выполняется бесконечно долго, NMPWAIT_USE_DEFAULT_WAIT - ожидание выполняется в течении периода времени, указанного при вызове функции CreateNamedPipe.

Возвращаемое значение: если канал доступен до истечения периода времени, то функция возвращает TRUE, иначе – FALSE.

Общая последовательность вызовов со стороны сервера:

```
hNp = CreateNamedPipe("\\\\.\\pipe\\my_pipe", ...);
while (... /* Цикл продолжается вплоть до завершения работы сервера.*/)
{
    ConnectNamedPipe(hNp, NULL); //подключаем клиента
    while (ReadFile(hNp, Request, ...) //пока он не закрыл соединение на своей
стороне принимаем сообщения
    {
        ...
        WriteFile(hNp, Response, ...); //шлем клиенту ответ
    }
    DisconnectNamedPipe(hNp); //отсоединяем клиента
}
CloseHandle(hNp);
```

Общая последовательность вызовов со стороны клиента:

```
WaitNamedPipe("\\.\\ServerName\\pipe\\my_pipe", NMPWAIT_WAIT_FOREVER);
//ждем разрешение сервера на соединение
hNp = CreateFile("\\.\\ServerName\\pipe\\my_pipe", ...); //подключаемся к
каналу
while (.../*Цикл выполняется до тех пор, пока не прекратятся запросы.*/
{
    WriteFile(hNp, Request, ...); //шлем запрос
    ...

    ReadFile(hNp, Response); //получаем ответ
}
CloseHandle (hNp); /* Разорвать соединение с сервером. */
```

Пример 2.1. Программа, обеспечивающая взаимодействие процессов посредством именованных каналов

Задание. Реализовать между одним клиентом и сервером обмен данными, вводимыми с консоли на стороне клиента и возвращаемыми сервером обратно до получения команды `exit`.

Описание программы

Программа-сервер (файл Master) создает именованный канал для двунаправленного использования и ожидает подключения программы-клиента (slave).

Проверяем, корректно ли произошло подключение, затем входим в цикл получения команд от "клиента" с последующими эхо-ответами. При появлении команды exit со стороны клиента, сервер завершает работу, закрывает канал. Клиент на своей стороне открывает канал, пишет в него и читает эхо-ответ. При вводе exit программа завершается.

Исходный код сервера *MASTER*:

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
int main()
{
    // Флаг успешного создания канала
    BOOL    fConnected;
    // Идентификатор канала Pipe
    HANDLE hNamedPipe;
    // Имя создаваемого канала Pipe
    LPCWSTR lpszPipeName = L"\\\\.\\pipe\\$MyPipe$";
    // Буфер для передачи данных через канал
    char    szBuf[512];
    // Количество байт данных, принятых через канал
    DWORD  cbRead;
    // Количество байт данных, переданных через канал
    DWORD  cbWritten;
    printf("Named pipe was creating \n");
    // Создаем канал Pipe, имеющий имя lpszPipeName
    hNamedPipe = CreateNamedPipe(
        lpszPipeName,           //имя канала
        PIPE_ACCESS_DUPLEX,    // режим котрытия канала - двунаправленный
                                //параметры режима pipe:
        PIPE_TYPE_MESSAGE     //Данные записываются в канал в виде потока сообщений
        | PIPE_READMODE_MESSAGE //Данные считываются в виде потока сообщений
        | PIPE_WAIT, // блокирующий режим
        PIPE_UNLIMITED_INSTANCES, //неограниченное кол-во "подключений" к каналу
        512, 512,              //кол-во байт входного и вызодного буферов
        0,                    //тайм-аут в 50 миллисекунд (по умолчанию)
        NULL );              // дескриптор безопасности по умолчанию
                                // Если возникла ошибка, выводим ее код и
                                // завершаем работу приложения
    if(hNamedPipe == INVALID_HANDLE_VALUE)
    {
        fprintf(stdout, "CreateNamedPipe: Error %ld\n",
            GetLastError());
        getch();
        return 0;
    }
    // Выводим сообщение о начале процесса создания канала
    fprintf(stdout, "Waiting for connect...\n");
    // Ожидаем соединения со стороны клиента
    fConnected = ConnectNamedPipe(hNamedPipe, // имя канала
        NULL); // overlapped=null
    // При возникновении ошибки выводим ее код
    if(!fConnected)
    {
        switch(GetLastError())
        {
            case ERROR_NO_DATA:
                fprintf(stdout, "ConnectNamedPipe: ERROR_NO_DATA");
        }
    }
}
```



```

    getch();
    CloseHandle(hNamedPipe);
    return 0;
break;
case ERROR_PIPE_CONNECTED:
    fprintf(stdout,
        "ConnectNamedPipe: ERROR_PIPE_CONNECTED");
    getch();
    CloseHandle(hNamedPipe);
    return 0;
break;
case ERROR_PIPE_LISTENING:
    fprintf(stdout,
        "ConnectNamedPipe: ERROR_PIPE_LISTENING");
    getch();
    CloseHandle(hNamedPipe);
    return 0;
break;
case ERROR_CALL_NOT_IMPLEMENTED:
    fprintf(stdout,
        "ConnectNamedPipe: ERROR_CALL_NOT_IMPLEMENTED");
    getch();
    CloseHandle(hNamedPipe);
    return 0;
break;
default:
    fprintf(stdout, "ConnectNamedPipe: Error %ld\n",
        GetLastError());
    getch();
    CloseHandle(hNamedPipe);
    return 0;
break;
}
CloseHandle(hNamedPipe);
getch();
return 0;
} // Выводим сообщение об успешном создании канала
fprintf(stdout, "\nConnected. Waiting for command...\n");

// Цикл получения команд из канала
while(1)
{ if(ReadFile(hNamedPipe, szBuf, 512, &cbRead, NULL))
    {
        // Выводим принятую команду на консоль
        printf("Received: <%s>\n", szBuf);
        // Если пришла команда "exit",
        // завершаем работу приложения
        if(!strcmp(szBuf, "exit"))
            break;

// отправляем эхо-ответ
if(!WriteFile(hNamedPipe, szBuf, strlen(szBuf) + 1, &cbWritten, NULL)) break;
    }
    else
    {getch();
      break;  }
}
CloseHandle(hNamedPipe);
return 0;}

```

Исходный код клиента slave программы

```

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

```

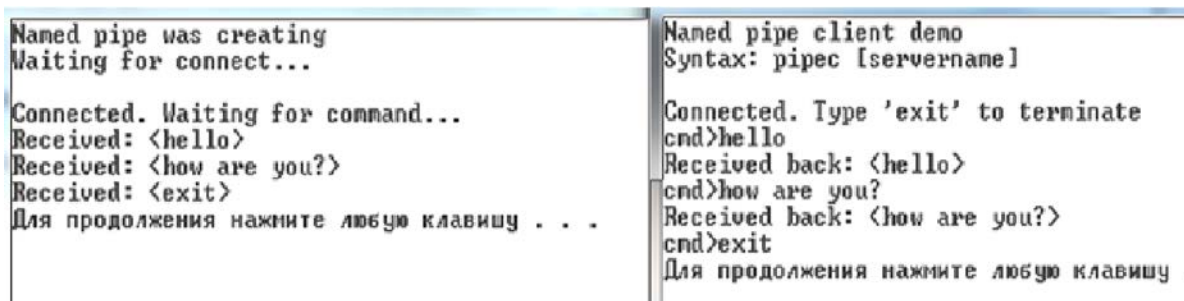
```

#include <conio.h>
DWORD main(int argc, char *argv[])
{ // Идентификатор канала Pipe
HANDLE hNamedPipe;
// Количество байт, переданных через канал
DWORD cbWritten;
// Количество байт, принятых через канал
DWORD cbRead;
// Буфер для передачи данных
char szBuf[256];
// Буфер для имени канала Pipe
LPCWSTR szPipeName=L"\\\\.\\pipe\\$MyPipe$";
printf("Named pipe client demo\n");
printf("Syntax: pipes [servername]\n");
// открываем handle нашего именованого pipe
hNamedPipe = CreateFile(
    szPipename, // pipe name
    GENERIC_READ | // read and write access
    GENERIC_WRITE,
    0, // no sharing
    NULL, // default security attributes
    OPEN_EXISTING, // opens existing pipe
    0, // default attributes
    NULL); // no template file
// Если возникла ошибка, выводим ее код и
// завершаем работу приложения
if(hNamedPipe == INVALID_HANDLE_VALUE)
{
    fprintf(stdout, "CreateFile: Error %ld\n",
        GetLastError());
    getch();
    return 0;
}
// Выводим сообщение о создании канала
fprintf(stdout, "\nConnected. Type 'exit' to terminate\n");
// Цикл обмена данными с серверным процессом
while(1)
{ // Выводим приглашение для ввода команды
    printf("cmd>");
    // Вводим текстовую строку
    gets(szBuf);
    // Передаем введенную строку серверному процессу
    // в качестве команды
    if(!WriteFile(hNamedPipe, szBuf, strlen(szBuf)+1, &cbWritten, NULL)) break;
    // Получаем эту же команду обратно от сервера
    if(ReadFile(hNamedPipe, szBuf, 512, &cbRead, NULL))
        printf("Received back: <%s>\n", szBuf);
    // В ответ на команду "exit" завершаем цикл
    // обмена данными с серверным процессом
    if(!strcmp(szBuf, "exit"))
        break;
}
// Закрываем идентификатор канала
CloseHandle(hNamedPipe);
return 0;
}

```

Результаты работы клиент-серверного приложения

Запускаем в одном окне сервер (на рис. слева). Он выводит сообщения о том, что канал создан и ожидает подключения клиента. Запускаем в другом окне клиента (справа).



```
Named pipe was creating
Waiting for connect...

Connected. Waiting for command...
Received: <hello>
Received: <how are you?>
Received: <exit>
Для продолжения нажмите любую клавишу . . .
```

```
Named pipe client demo
Syntax: pipec [servername]

Connected. Type 'exit' to terminate
cmd>hello
Received back: <hello>
cmd>how are you?
Received back: <how are you?>
cmd>exit
Для продолжения нажмите любую клавишу .
```

Рис.20. Результаты работы приложения с именованным каналом

Затем на стороне сервера мы получаем уведомление о том, что создано новое подключение, теперь мы можем получать сообщения от клиента.

Клиент после запуска выводит уведомления о том, что он подключился к серверу и ожидает ввода команд в строке **cmd>**. Вводимые в командной строке символы он пересылает серверу и сразу выводит возвращенный эхо-ответ от него. Для завершения сеанса обмена следует ввести зарезервированную команду **exit** на стороне клиента, после ее доставки серверу, он завершает работу.

Сервер работает с одним клиентом, поэтому ему не нужно вызывать функции отсоединения клиента (по завершению сервера, клиент тоже завершается).

Пример 2.2. Программа, обеспечивающая взаимодействие процессов посредством именованных каналов – аналогичная программа с эхо-сервером, но с множеством клиентов и принудительной блокировкой обмена до завершения каждой операции.

Задание. Реализовать между сервером и множеством клиентов обмен данными, вводимыми с консоли на стороне клиента и возвращаемыми сервером обратно до получения команды **exit**.

Описание программы

Сервер, как и ранее, создает все необходимые ресурсы и переходит в состояние ожидания соединений. Именованный канал создается для чтения и записи. Передача происходит сообщениями, функции передачи и приема блокируются до их окончания.

Клиент после соединения с сервером начинает чтение сообщений с консоли, пока не встретит слово «exit». По данному слову и клиент и сервер завершают свою работу.

Обратите внимание на использование функции **WaitNamedPipe**, а также на возможность использования количества экземпляров каналов, равного количеству потенциальных клиентов.

Исходный код сервера:

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

#define SIZE_OF_BUF 1000

int main()
{
    HANDLE hNamedPipe; // Идентификатор канала Pipe
    // Имя создаваемого канала Pipe
    LPSTR lpszPipeName = "\\.\pipe\\$MyPipe$";
    char buf[SIZE_OF_BUF]; // Буфер для передачи данных через канал
    DWORD readbytes, writebytes; // число байт прочитанных и переданных

    printf("Server is started. Try to create named pipe\n");

    // Создаем канал Pipe, имеющий имя lpszPipeName
    hNamedPipe = CreateNamedPipe(
        lpszPipeName, // имя канала
        PIPE_ACCESS_DUPLEX, // доступ и на чтение и на запись
        PIPE_TYPE_MESSAGE | PIPE_READMODE_MESSAGE | PIPE_WAIT,
        // передача сообщений (как чтение, так и запись)
        5, // максимальное число экземпляров каналов равно 5 (число клиентов)
        SIZE_OF_BUF, SIZE_OF_BUF, 5000, NULL); // размеры выходного и
    // входного буферов канала, 5 секунд - длительность для функции
    // WaitNamedPipe

    if(hNamedPipe == INVALID_HANDLE_VALUE) // Если возникла ошибка,
    // выводим ее код и завершаем работу приложения
    {
        fprintf(stdout, "CreateNamedPipe: Error %ld\n", GetLastError());
        getchar();
        return 1000;
    }
    printf("Named pipe created successfully\n");
    // Выводим сообщение о начале процесса создания канала
    printf("waiting for connect\n");
    // Ожидаем соединения со стороны клиента
    if(!ConnectNamedPipe(hNamedPipe, NULL))
    {
        // При возникновении ошибки выводим ее код
        printf("error with function ConnectNamedPipe\n");
    }
}
```

```

    getchar();
    CloseHandle(hNamedPipe);
    return 1001;
}

// Выводим сообщение об успешном создании канала
fprintf(stdout,"Client connected\n");
// Цикл получения команд через канал
while(1)
{
    // Получаем очередную команду через канал Pipe
    if(ReadFile(hNamedPipe, buf, SIZE_OF_BUF, &readbytes, NULL))
    {
        // Посылаем эту команду обратно клиентскому приложению
        if(!WriteFile(hNamedPipe, buf, strlen(buf) + 1, &writebytes, NULL))
            break;
        // Выводим принятую команду на консоль
        printf("Get client msg: %s\n", buf);
        // Если пришла команда "exit",
        // завершаем работу приложения
        if(!strcmp(buf, "exit",4))
            break;
    }
    else
    {
        fprintf(stdout,"ReadFile: Error %ld\n",GetLastError());
        getchar();
        break;
    }
}

CloseHandle(hNamedPipe);
printf("server is ending\n press any key\n");
getchar();
return 0;
}

```

Исходный код клиента:

```

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

#define SIZE_OF_BUF 1000

DWORD main(int argc, char *argv[])
{
    HANDLE hNamedPipe; // Идентификатор канала Pipe
    DWORD readbytes,writebytes; // количество байт принятых и
//переданных в канал
    char buf[SIZE_OF_BUF]; // Буфер для передачи данных
    printf("Client is started\n Try to use WaitNamedPipe\n");
    //"ожидает" пока освободится канал

    if(!WaitNamedPipe("\\\\.\\pipe\\$$MyPipe$$",NMPWAIT_WAIT_FOREVER))

```

```

{
    printf("pipe wasn't created\n getlasterror = %d",
GetLastError());
    getchar();
    return 1000;
}
// Создаем канал с процессом-сервером
hNamedPipe = CreateFile("\\\\.\\pipe\\$$MyPipe$$", GENERIC_READ
| GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, NULL);

// Если возникла ошибка, выводим ее код и
// завершаем работу приложения
if(hNamedPipe == INVALID_HANDLE_VALUE)
{
    fprintf(stdout, "CreateFile: Error %ld\n", GetLastError());
    getchar();
    return 1001;
}
// Выводим сообщение о создании канала
printf("successfully connected\n input message\n");
// Цикл обмена данными с серверным процессом
while(1)
{
    // Выводим приглашение для ввода команды
    printf("cmd>");

    // Вводим текстовую строку
    fgets(buf, SIZE_OF_BUF, stdin);
    // Передаем введенную строку серверному процессу
    // в качестве команды
    if(!WriteFile(hNamedPipe, buf, strlen(buf) + 1, &writebytes,
NULL))
    {
        printf("connection refused\n");
        break;
    }
    // Получаем эту же команду обратно от сервера
    if(ReadFile(hNamedPipe, buf, SIZE_OF_BUF, &readbytes, NULL))
        printf("Received from server: %s\n", buf);

    // Если произошла ошибка, выводим ее код и
    // завершаем работу приложения
    else
    {
        fprintf(stdout, "ReadFile: Error %ld\n", GetLastError());
        getchar();
        break;
    }
    // В ответ на команду "exit" завершаем цикл
    // обмена данными с серверным процессом
    if(!strncmp(buf, "exit", 4))
        break;
}
}

```

```

// Закрываем идентификатор канала
CloseHandle(hNamedPipe);
printf("client is ending\n Press any key\n");
getchar();
return 0;
}

```

Результаты работы программы представлены на рис. 21 и соответствуют эксперименту с одним клиентом.

```

Server is started. Try to create named pipe
Named pipe created successfully
waiting for connect
Client connected
Get client msg: string1
Get client msg: string2
Get client msg: very long string
Get client msg: exit
server is ending
press any key

Client is started
Try to use WaitNamedPipe
successfully connected
input message
cmd>string1
Received from server: string1

cmd>string2
Received from server: string2

cmd>very long string
Received from server: very long string

cmd>exit
Received from server: exit

client is ending
Press any key

```

Рисунок 21. Результат работы программы

Самостоятельно организуйте эксперимент с множеством клиентов, проанализируйте взаимодействие, если необходимо, введите средства синхронизации, обоснуйте их выбор, представьте алгоритм и диаграмму взаимодействия, используйте для этого инструментарий среды программирования.

Сетевая передача данных с помощью именованных каналов

Именованные каналы позволяют осуществлять обмен между процессами, выполняющимися на разных компьютерах в сети. Для этого необходимо выполнить определенный ряд условий и настроек. Но сетевые именованные каналы не являются промышленным стандартом и используются в этом качестве скорее как исключение. В ОС семейства Windows это возможно, в отличие, например, от Unix-подобных систем, где обмен данными осуществляется через ядро.

Рассмотрим эту возможность Windows на примере.

Пример 2.3. Модифицируем приложение из предыдущего примера (2.2) для сетевого обмена информацией.

Для совместной работы компьютеры нужно подсоединить к одной домашней группе. Так же необходимо установить поле DACL (Discretionary Access Control List) защиты объекта в NULL (разрешение всем пользователям и группам доступа к объекту). Параметры защиты именованного канала задаются с помощью структуры `SECURITY_ATTRIBUTES`, которая указывается последним параметром в функции `CreateNamedPipe`.

Измененный код сервера:

```
...
printf("Server is started. Try to create named pipe\n");
/*****
*****/

// Создание SECURITY_ATTRIBUTES и SECURITY_DESCRIPTOR объектов
SECURITY_ATTRIBUTES sa;
SECURITY_DESCRIPTOR sd;
// Инициализация SECURITY_DESCRIPTOR значениями по-умолчанию
if (InitializeSecurityDescriptor(&sd,
SECURITY_DESCRIPTOR_REVISION) == 0)
{
    printf("InitializeSecurityDescriptor failed with error %d\n",
GetLastError());
    return 50000;
}
// Установка поля DACL в SECURITY_DESCRIPTOR в NULL
if (SetSecurityDescriptorDacl(&sd, TRUE, NULL, FALSE) == 0)
{
    printf("SetSecurityDescriptorDacl failed with error %d\n",
GetLastError());
    return 50001;
}
// Установка SECURITY_DESCRIPTOR в структуре SECURITY_ATTRIBUTES
sa.nLength = sizeof(SECURITY_ATTRIBUTES);
sa.lpSecurityDescriptor = &sd;
sa.bInheritHandle = FALSE; //запрещение наследования
hNamedPipe = CreateNamedPipe(
    lpzPipeName, //имя канала
    PIPE_ACCESS_DUPLEX, //доступ и на чтение и на запись
    PIPE_TYPE_MESSAGE | PIPE_READMODE_MESSAGE | PIPE_WAIT,
//передача сообщений (как чтение, так и запись)
    5, //максимальное число экземпляров каналов равно 5 (число
клиентов)
    SIZE_OF_BUF, SIZE_OF_BUF, 5000, &sa); //размеры выходного и
входного буферов канала, 5 секунд - длительность для функции
WaitNamedPipe
```



```

//*****
*****

if(hNamedPipe == INVALID_HANDLE_VALUE)
...

```

Клиент в качестве имени канала использует строку следующего вида:

```
char * pipename="\\\\server_name\\pipe\\МуPipe";
```

В качестве server_name можно использовать либо IP адрес, либо имя компьютера.

Измененный код клиента:

```

...
printf("Client is started\n Try to use WaitNamedPipe\n");

char * pipename="\\\\192.168.0.5\\pipe\\МуPipe"; //адрес сервера и
//имя канала
//"ожидаем" пока освободится канал
if(!WaitNamedPipe(pipename,NMPWAIT_WAIT_FOREVER))
{
    printf("pipe wasn't created\n GetLastError = %d",
GetLastError());
    getchar();
    return 1000;
}
...

```

Результаты работы программы представлены на рис. 22 и 23 для клиента и сервера соответственно.

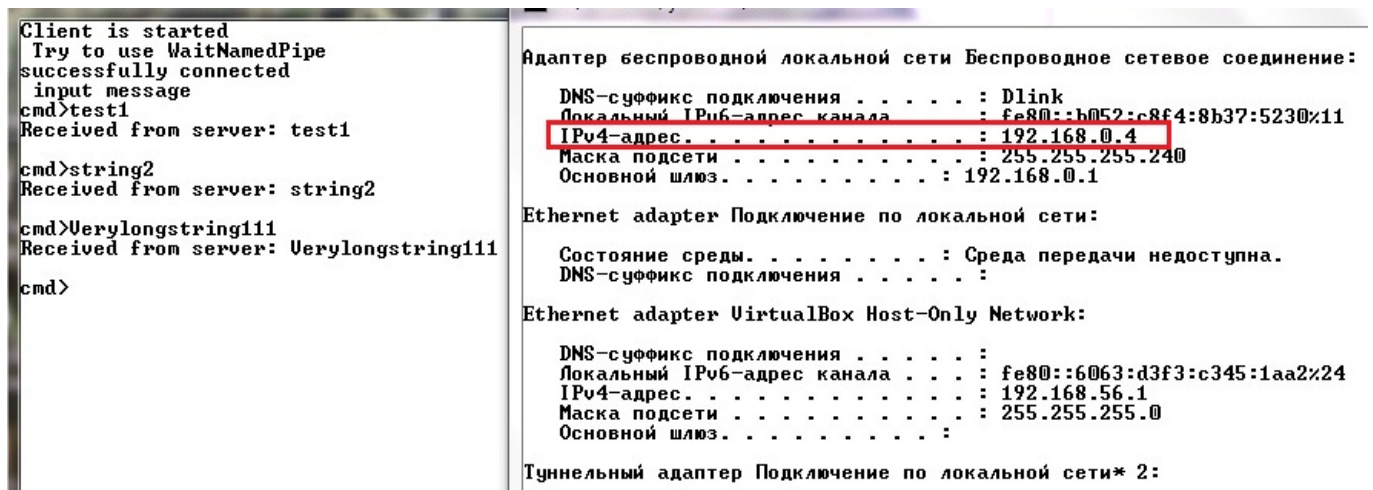


Рисунок 22. Использование именованных каналов. Результат запуска на машине клиента

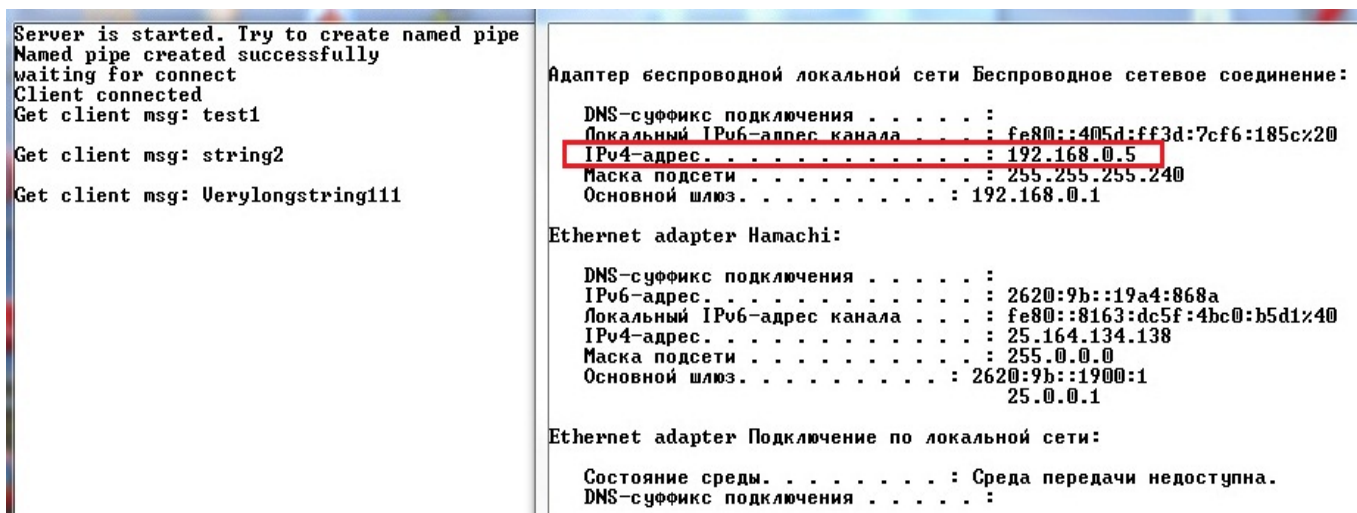


Рисунок 23. Использование именованных каналов. Результат запуска на машине сервера

Прямоугольниками выделены адрес клиента (рис. 22) и адрес сервера (рис. 23).

Самостоятельно

- проведите эксперимент на виртуальных машинах, опишите отладочный комплекс – виртуальную сеть; представьте результаты, используя системные утилиты и результаты работы ваших программ;
- проведите эксперимент в лаборатории, опишите реальную сеть, в которой проводится эксперимент, представьте результаты. Какие настройки потребовалось изменить?

3. Сокеты

Информация об используемом средстве IPC

Возможность взаимодействия с другими системами обеспечивается в Windows поддержкой сокетов (sockets) Windows Sockets — совместимого и почти точного аналога сокетов Berkeley Sockets, де-факто играющих роль промышленного стандарта.

Winsock API разрабатывался как расширение Berkley Sockets API для среды Windows и поэтому поддерживается всеми системами Windows.

Winsock API поддерживается библиотекой DLL (WS2_32.DLL), для получения доступа к которой следует подключить к программе библиотеку WS_232.LIB. Эту DLL следует инициализировать с помощью нестандартной, специфической для Winsock функции WSAStartup, которая должна быть первой из функций

Winsock, вызываемых программой. Когда необходимость в использовании функциональных возможностей Winsock отпадает, следует вызывать функцию WSACleanup.

```
int WSASStartup(WORD wVersionRequired, LPWSADATA ipWSAData);
```

WORD wVersionRequired - указывает старший номер версии библиотеки DLL, который требуется и который можно использовать. Младший байт параметра указывает основной номер версии, а старший байт — дополнительный;

LPWSADATA ipWSAData - указатель на структуру WSADATA, которая возвращает информацию о конфигурации DLL, включая старший доступный номер версии.

Возвращаемое значение: ненулевое значение, если запрошенная вами версия данной DLL не поддерживается.

По окончании работы программы, а также в тех случаях, когда необходимости в использовании сокетов больше нет, следует вызывать функцию WSACleanup, чтобы библиотека WS_32.DLL, обслуживающая сокет, могла освободить ресурсы, распределенные для этого процесса.

После инициализации библиотеки сокетов можно использовать **стандартные функции работы с сокетами.**

Первой из них является **функция создания сокета:**

```
SOCKET socket(int af, int type, int protocol);
```

int af – обозначает семейство адресов или протокол (для указания протокола TCP/IP необходимо указать PF_INET или AF_INET);

int type – указывает тип взаимодействия: SOCK_STREAM – потоковое (соответствует протоколу TCP в стеке TCP/IP), SOCK_DGRAM – дейтаграммное (соответствует протоколу UDP в стеке TCP/IP);

int protocol – параметр является лишним (равен 0), если параметр af установлен как AF_INET.

Возвращаемое значение: в случае неудачного завершения функция вернет INVALID_SOCKET.

Взаимодействие в сети осуществляется между клиентом и сервером. Клиент посылает серверу некоторый запрос. Сервер обрабатывает запрос и шлет ответ.

Сам по себе сокет – это оконечная точка соединения, которая идентифицируется 4 значениями: IP адрес отправителя, порт отправителя, IP адрес получателя, порт получателя.

Порт – идентификатор процесса в ОС с точки зрения сетевого взаимодействия. Порт напрямую связан с протоколом. Например, в ОС могут быть два процесса с одинаковым номером порта, но использующие при этом разные протоколы. В качестве номера порта желательно выбирать значение большее 1024 (т.к. это зарезервированные порты).

Рассмотрим **функции, которые должны вызываться сервером.**

Созданный сокет (после вызова функции `socket`) **ассоциируется с семейством адресов**, а не с конкретным адресом.

До того как сокет сможет принять входящие соединения, он должен быть связан с адресом. Для этого используется **функция связи с адресом:**

```
int bind(SOCKET s, const struct sockaddr *saddr, int namelen);
```

SOCKET s – несвязанный сокет, возвращенный функцией `socket`;

*struct sockaddr *saddr* – специальная структура, которая задает протокол, адрес и порт;

int namelen – размер второго параметра функции.

Возвращаемое значение: в случае успешного выполнения функция вернет 0, иначе – `SOCKET_ERROR`.

Структура **sockaddr** определяется следующим образом:

```
struct sockaddr {
    u_short sa_family; //обозначает протокол
    char sa_data[14]; //заполнение зависит от протокола
};
```

Internet-версией структуры **sa_data** является структура **sockaddr_in:**

```
struct sockaddr_in {
    short sin_family; /* AF_INET */
    u_short sin_port; //номер порта
    struct in_addr sin_addr; /* 4-байтовый IP-адрес */
    char sin_zero[8];
};
```

Номер порта хранится **в сетевом порядке байт**, т.е. старший байт размещается в старшей позиции справа. Для корректной установки порта необходимо пользоваться функцией:

```
uint16_t htons(uint16_t hostshort);
```

Данная функция расшифровывается так: «host to network short». Она производит преобразование порядка записи байт, меняя порядок «хоста» в «сетевой».

Структура `sin_addr` содержит подструктуру `s_addr`, которая заполняется 4-байтовым IP-адресом, например 127.0.0.1. Он указывает систему, запрос на образование соединения которой должен быть принят. Обычно удовлетворяются запросы любых систем, поэтому следует использовать значение `INADDR_ANY`.

Для перевода адреса IP из текстового вида в число необходимо воспользоваться функцией:

```
in_addr_t inet_addr(const char *cp);
```

Возвращаемое значение: -1 – при ошибке, иначе – значение адреса.

Пример заполнения адреса:

```
sa.sin_addr.s_addr = inet_addr("192 .13.12.1");
```

О связанном сокете, для которого определены протокол, номер порта и IP-адрес, иногда говорят как об **именованном сокете (named socket)**.

Для того, чтобы сокет мог принимать соединения, его необходимо сделать **прослушивающим**. Для этого существует специальная **функция**:

```
int listen(SOCKET s, int nQueueSize);
```

int nQueueSize – максимальное число запросов на соединение, которые будут помещены в очередь.

Возвращаемое значение: в случае успеха функция вернет 0, иначе – `SOCKET_ERROR`.

Для приема соединений используется функция:

```
SOCKET accept(SOCKET s, LPSOCKADDR lpAddr, LPINT lpAddrLen);
```

SOCKET s – слушающий сокет;

LPSOCKADDR lpAddr – структура адреса, заполненная информацией о клиентской системе;

LPINT lpAddrLen – размер структуры адреса.

Возвращаемое значение: новый сокет (уже не прослушивающий), связанный с клиентом. Отсылка данных в данный сокет приведет к передаче их клиенту.

Вызов функции `accept` является блокируемым: пока не придет соединение, функция не вернет управление.

Для закрытия сокета (для удаления всех связанных с ним ресурсов) используется функция:

```
closesocket(SOCKET s)
```

Сервер должен закрывать прослушивающий сокет только по окончании работы.

Общий *порядок вызовов функций сервера*:

```
struct sockaddr_in SrvSAddr; /* Адресная структура сервера. */
struct sockaddr_in ConnectAddr;
SOCKET SrvSock, sockio;
...
SrvSock = socket(AF_INET, SOCK_STREAM, 0); //получение сокета
SrvSAddr.sin_family = AF_INET;
SrvSAddr.sin_addr.s_addr = htonl(INADDR_ANY);
SrvSAddr.sin_port = htons(SERVER_PORT);
bind(SrvSock, (struct sockaddr *)&SrvSAddr, sizeof SrvSAddr);
//связывание сокета с адресом
listen(SrvSock, 5); //установка его прослушивающим
AddrLen = sizeof(ConnectAddr);
sockio = accept(SrvSock, (struct sockaddr *) &ConnectAddr, &AddrLen);
//прием соединения от клиентов
... Получение запросов и отправка ответов ...
closesocket(sockio); //закрытие сокета
```

Далее следует рассмотреть *клиентские функции сокета*.

Первая из таких функций позволяет клиенту подключиться к серверу:

```
int connect(SOCKET s, LPSOCKADDR lpName, int nNameLen);
```

SOCKET s – сокет, созданный с помощью вызова socket. При этом сокет не обязательно должен быть связанным. Если он не был связан, система сама установит ему адрес и порт;

LPSOCKADDR lpName – указатель на структуру sockaddr_in, инициализированную значениями номера порта и IP-адреса системы с сокетом, связанным с указанным портом, который находится в состоянии прослушивания;

int nNameLen – размер структуры sockaddr_in.

Возвращаемое значение: 0 – успешное завершение, SOCKET_ERROR – ошибка.

Клиент производит подключение к серверу с помощью следующих вызовов:

```
SOCKET ClientSock;
...
ClientSock = socket(AF_INET, SOCK_STREAM, 0); //создание сокета
memset(&ClientSAddr, 0, sizeof(ClientSAddr));
ClientSAddr.sin_family = AF_INET;
```

```
ClientSAddr.sin_addr.s_addr = inet_addr(argv[1]);
ClientSAddr.sin_port = htons(SERVER_PORT);
ConVal = connect(ClientSock, (struct sockaddr *)&ClientSAddr,
sizeof(ClientSAddr)); //подсоединение к серверу
```

Стоит отметить, что *реальная установка соединения происходит только в соответствующих протоколах*. Например, в протоколе UDP при вызове функции connect никакого соединения установлено не будет (произойдет **только связывание сокета с адресами и портами сервера и клиента**).

После установки соединения, появляется возможность использовать функции приема и передачи данных. Для **передачи данных** используется:

```
int send(SOCKET s, const char * lpBuffer, int nBufferLen, int nFlags);
```

*const char * lpBuffer* – указатель на буфер, из которого передаются данные;

int nBufferLen – размер передаваемых данных в байтах;

int nFlags – флаги. Например, флаг MSG_PEEK позволяет прочитать данные из сокета без удаления их оттуда, а флаг MSG_WAITALL – заблокирует процесс в функции recv, пока не будет прочитано заданное число байт.

Возвращаемое значение – число фактически переданных байт. Стоит отметить, что не обязательно, что за один вызов будут переданы все байты (если не был установлен специальный флаг).

Для приема данных используется функция:

```
int recv(SOCKET s, char *buf, int nBufferLen, int nflags);
```

Параметры и возвращаемое значение которой аналогичны функции send.

Необходимо отметить, что сокеты бывают **блокируемыми** и **неблокируемыми**. По *умолчанию*, сокеты являются *блокируемыми*. Например, если поток вызвал функцию read на сокете, а данных в буфере сокета нет, то функция read не вернет управление, пока данные не появятся.

Так же для работы с сокетами можно использовать функции : select, poll, epoll,..., которые позволяют получать уведомления о событиях, происходящих на наблюдаемых сокетах.

Пример 3.1. Программа локального обмена сокетами с использованием потокового протокола с установлением соединения (TCP в стеке TCP/IP).

Для потоковых протоколов (к которым относится протокол TCP в стеке TCP/IP) необходимо применять средства, позволяющие определить границы

сообщений в передаваемых данных, так как данный вид протоколов имеет дело с доставкой только потока байт (при этом гарантируется порядок доставки).

В качестве функций, обеспечивающих передачу по разделителям, используются:

- `int recvn(SOCKET fd, char *bp, size_t len);` - функция, которая читает из сокета заданное число байт;
- `int sendn(SOCKET s, char* buf, int lenbuf, int flags);` - функция, которая передает в сокет заданное число байт;
- `int sendLine(int sock, char* str);` - функция, которая добавляет к исходным данным символ-разделитель сообщений и передает сообщение функции `sendn`;
- `int recvLine(SOCKET s, char* buffer, int buffSize);` - функция, которая читает буфер сокета до разделителя и полученную строку возвращает (разделитель исключается).

Исходные коды перечисленных функций представлены ниже, после кода сервера и клиента.

Исходный код сервера:

```
int main(void)
{
    //используется для инициализации библиотеки сокетов
    WSADATA WSStartData;
    //Инициализация WinSock и проверка его запуска
    if (WSAStartup(MAKEWORD(2, 0), &WSStartData) != 0)
    {
        printf("WSAStartup failed with error: %d\n", GetLastError());
        return 100;
    }
    //создание сокета
    SOCKET server_socket;
    //по умолчанию используется протокол tcp
    printf("Server is started.\nTry to create socket -----
");
    if((server_socket = socket( AF_INET, SOCK_STREAM, 0 ))
==INVALID_SOCKET) //протокол в домене Internet, надежный с
//установлением соединения
    {printf("error with creation socket. GetLastError=
%d\n",GetLastError());
        return 1000;
    }
    printf("CHECK\n");
    //Привязывание сокета конкретному IP и номеру порта
    struct sockaddr_in sin;
```



```

    sin.sin_addr.s_addr=inet_addr("127.0.0.1"); // используем локальную
//машину
    sin.sin_port=htons(7500); // может быть любым кроме зарезервированных
    sin.sin_family=AF_INET;

    printf("Try to bind socket -----");
    if ( bind( server_socket, (struct sockaddr *)&sin, sizeof(sin) ) !=0)
    {
        printf("error with bind socket. GetLastError= %d\n",GetLastError());
        return 1001;
    }
    printf("CHECK\n");
    //делаем сокет прослушиваемым
    printf("Try to set socket listening -----");
    if(listen(server_socket,5 )!=0)
    {
        printf("error with listen socket. GetLastError= %d\n",GetLastError());
        return 1002;
    }
    printf("CHECK\n");
    printf("Server starts listening\n");
    //ждем клиента. Создаем пустую структуру, которая будет содержать
//параметры сокета, иницирующего соединение
    struct sockaddr_in from;
    int fromlen=sizeof(from);
    // начинаем "слушать" входящие запросы на подключение
    SOCKET client_socket=accept(server_socket,(structsockaddr*)&from,&fromlen);
    if(client_socket==INVALID_SOCKET)
    {
        printf("error with accept socket. GetLastError= %d\n",GetLastError(
        return 1003;
    }

    printf("get client with IP= %s, port = %d\n",
inet_ntoa(from.sin_addr),ntohs(from.sin_port));

    char buf[SIZE_OF_BUF]; //буфер приема и передачи сообщения
    int readbytes; //число прочитанных байт
    while(1)
    {
        if((readbytes=recvLine(client_socket,buf,SIZE_OF_BUF))==0)
        {
            printf("Connection refused\n");
            break;
        }
        else if(readbytes==-1)
        {
            printf("buf is small\n");
            return 2000;
        }
    }

```

```

    printf("get msg from client \"%s\" with size= %d\n",buf,readbytes);
    sendLine(client_socket,buf);
    //sendn(client_socket,buf,readbytes,0);    //отправка сообщения
//обратно клиенту

    if(strncmp(buf,"exit",4)==0)
        break;
}

closesocket(client_socket);
closesocket(server_socket);
return 0;
}

```

Исходный код клиента:

```

int main (void)
{
    //используется для инициализации библиотеки сокетов
    WSADATA WSStartData;
    //Инициализация WinSock и проверка его запуска
    if (WSAStartup(MAKEWORD(2, 0), &WSStartData) != 0)
    {
        printf("WSAStartup failed with error: %d\n", GetLastError());
        return 100;
    }
    int er_code=0;
    // инициализация клиентского сокета
    printf("Client is started.\nTry to create socket\n");
    int client_socket = socket( AF_INET, SOCK_STREAM, 0 );
    printf("socket created successfully\n");
    struct sockaddr_in sin;
    sin.sin_addr.s_addr=inet_addr("127.0.0.1");
    sin.sin_port=htons(7500);
    sin.sin_family=AF_INET;
    // устанавливаем TCP-соединение
    printf("try to connect to server\n");
    if(connect(client_socket, (struct sockaddr *) &sin,sizeof(sin))!=0)
    {
        printf("connect failed with error: %d\n", er_code);
        return SOCKET_ERROR;
    }
    printf("Client connected sucessfully\nEnter msg to send\n-----
    -----\n");

    char buf[SIZE_OF_BUF];    //буфер для приема и передачи сообщений
    while(1)
    {
        fgets(buf,SIZE_OF_BUF,stdin);
        printf("client sended msg: %s",buf);
        sendLine(client_socket,buf);
        recvLine(client_socket,buf,SIZE_OF_BUF);
    }
}

```

```

        printf("get msg from serv:
\\\"%s\\\"\\n*****\\n\",buf);

    }
    // заканчиваем работу с сокетом клиента
    closesocket(client_socket);
    return 0;
}

```

Исходные коды функций, обеспечивающих передачу по разделителям:

```

int sendn(SOCKET s, char* buf, int lenbuf, int flags) {
    int bytesSended = 0; //
    int n; //

    while (bytesSended < lenbuf) //шлем данные, пока количество
переданных байт не сравняется с необходимым
{
        n = send(s, buf + bytesSended, lenbuf - bytesSended, flags);
        if (n < 0) {
            //printf("Error with send in sendn\\n");
            break;
        }
        bytesSended += n;
    }
    return (n == -1 ? -1 : bytesSended);
}

int recvLine(SOCKET sock, char* buffer, int buffSize) { //функция приема
сообщения
    char* buff = buffer; //указатель на начало внешнего буфера
    char* currPosPointer; //указатель для работы со временным буфером
    int count = 0; //число прочитанных символов (без удаления из
буфера сокета)
    char tempBuf[100]; //временный буфер для приема
    char currChar; //текущий анализируемый символ (ищем
разделитель)

    int tmpcount = 0;
    while (--buffSize > 0) //пока во внешнем буфере (куда пишется сообщение) есть
//место
    {
        if (--count <= 0) {

            recvn(sock, tempBuf, tmpcount);
            count = recv(sock, tempBuf, sizeof (tempBuf), MSG_PEEK);
            if (count <= 0) {
                return count;
            }
            currPosPointer = tempBuf;

            tmpcount = count;
        }
    }
}

```

```

        currChar = *currPosPointer++;
        *buffer++ = currChar;
        if (currChar == '\n') {
            *(buffer - 1) = '\0';
            recvn(sock, tempBuf, tmpcount - count + 1);
            return buffer - buff-1;
        }
    }
    return -1;
}

int recvn(SOCKET fd, char *bp, size_t len) {
    return recv(fd, bp, len, MSG_WAITALL);
}

int sendLine(int sock, char* str) {
    char tempBuf[MAX_STR_LEN];
    strcpy(tempBuf, str);
    if(tempBuf[strlen(tempBuf)-1]!='\n')
        strcat(tempBuf, "\n");
    return sendn(sock, tempBuf, strlen(tempBuf), 0);
}

```

Результаты выполнения представлены на рис. 24.

<pre> Server is started. Try to create socket -----CHECK Try to bind socket -----CHECK Try to set socket listening -----CHECK Server starts listening get client with IP= 127.0.0.1, port = 57590 get msg from client "teststring1" with size= 11 get msg from client "very long string 1" with size= 18 </pre>	<pre> Client is started. Try to create socket socket created successfully try to connect to server Client connected successfully Enter msg to send ----- teststring1 client sent msg: teststring1 get msg from serv: "teststring1" ***** very long string 1 client sent msg: very long string 1 get msg from serv: "very long string 1" ***** </pre>
---	--

Рисунок 24. Результат обмена с использованием сокетов

Самостоятельно

модифицировать программу для локального обмена с множеством клиентов и с доступом к общему ресурсу.

Сетевая передача данных с помощью сокетов

В код программ клиента и сервера необходимо внести **незначительные изменения**, поскольку сокет изначально предназначен для удаленного взаимодействия.

Слушающий сокет сервера необходимо привязывать к адресу `INADDR_ANY`, чтобы он мог принимать соединения с любых адресов.

Клиенту необходимо указать IP адрес компьютера, на котором запущен клиент. Чтобы узнать IP адрес компьютера, можно вызвать в консоли команду **ipconfig**.

Работоспособность *слушающего сокета* можно определить с помощью команды **netstat** с флагами:

- -p – для указания протокола, по которому необходимо вывести информацию;
- -a – для получения всех соединений.

Изменения в коде сервера:

```
...
printf("CHECK\n");
//Привязывание сокета конкретному IP и номеру порта
struct sockaddr_in sin;
sin.sin_addr.s_addr=INADDR_ANY; //разрешаем подключение клиентов с
//любыми адресами
sin.sin_port=htons(7500); // может быть любым кроме зарезервированных
sin.sin_family=AF_INET;
...
```

Изменения в коде клиента:

```
...
// инициализация клиентского сокета
printf("Client is started.\nTry to create socket\n");
int client_socket = socket( AF_INET, SOCK_STREAM, 0 );
printf("socket created successfully\n");
struct sockaddr_in sin;
sin.sin_addr.s_addr=inet_addr("10.1.208.10"); //устанавливаем адрес
//сервера
sin.sin_port=htons(7500);
sin.sin_family=AF_INET;
...
```

Результаты запуска сервера и клиента представлены на рис. 25 и 26.

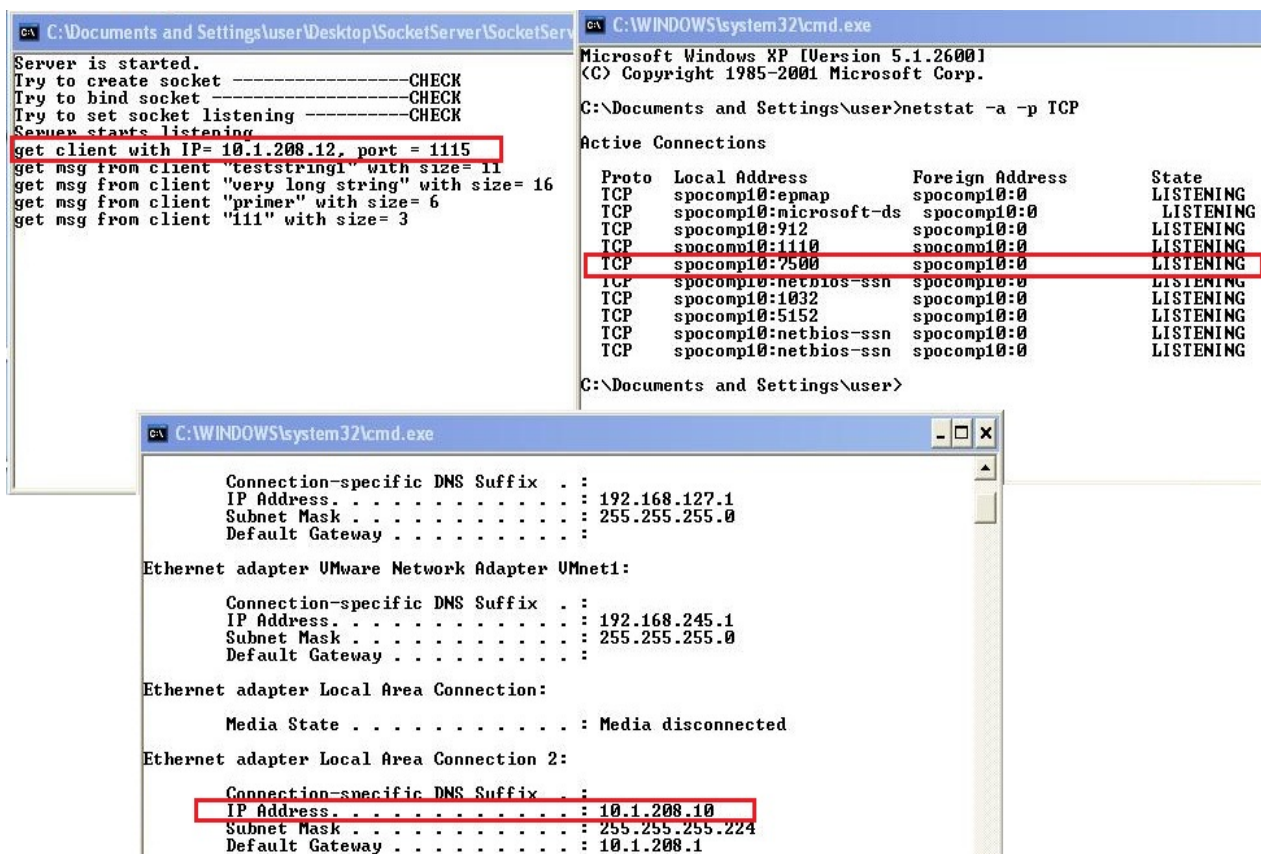


Рис. 25. Использование сокетов. Результат запуска на компьютере сервера

Левая консоль связана с процессом-сервером. В ней отображается начальная инициализация сервера, адрес и порт подключенного клиента (выделено прямоугольником). Каждое сообщение, посылаемое клиентом, выводится на стандартный вывод и пересылается обратно клиенту.

Правая консоль – отображает информацию, выводимую утилитой **netstat**. Прямоугольником выделена строка, соответствующая слушающему сокету сервера.

Нижняя консоль – отображает информацию, выводимую утилитой **ipconfig**. Прямоугольником выделен адрес физического интерфейса, к которому присоединяется клиент (остальные интерфейсы относятся к виртуальной машине).

```

C:\Documents and Settings\user\Desktop\Socket C:\WINDOWS\system32\cmd.exe
Client is started.
Try to create socket
socket created successfully
try to connect to server
Client connected successfully
Enter msg to send
-----
teststring1
client sent msg: teststring1
get msg from serv: "teststring1"
*****
very long string
client sent msg: very long string
get msg from serv: "very long string"
*****
primer
client sent msg: primer
get msg from serv: "primer"
*****
111
client sent msg: 111
get msg from serv: "111"
*****
TCP SPOComp12:epmap SPOComp12:0 LISTENING
TCP SPOComp12:microsoft-ds SPOComp12:0 LISTENING
TCP SPOComp12:912 SPOComp12:0 LISTENING
TCP SPOComp12:1110 SPOComp12:0 LISTENING
TCP SPOComp12:nethios-ssn SPOComp12:0 LISTENING
TCP SPOComp12:1025 SPOComp12:0 LISTENING
TCP SPOComp12:nethios-ssn SPOComp12:0 LISTENING
TCP SPOComp12:nethios-ssn SPOComp12:0 LISTENING

C:\Documents and Settings\user>netstat -p TCP -a

Active Connections

Proto Local Address Foreign Address State
TCP SPOComp12:epmap SPOComp12:0 LISTENING
TCP SPOComp12:microsoft-ds SPOComp12:0 LISTENING
TCP SPOComp12:912 SPOComp12:0 LISTENING
TCP SPOComp12:1110 SPOComp12:0 LISTENING
TCP SPOComp12:nethios-ssn SPOComp12:0 LISTENING
TCP SPOComp12:1115 10.1.208.10:7500 ESTABLISHED
TCP SPOComp12:1025 SPOComp12:0 LISTENING
TCP SPOComp12:nethios-ssn SPOComp12:0 LISTENING
TCP SPOComp12:nethios-ssn SPOComp12:0 LISTENING

```

Рис. 26. Использование сокетов. Результат запуска на компьютере клиента

Левая консоль связана с процессом клиента. В нем пользователь печатает сообщения для передачи серверу.

Правая консоль – результат вызова утилиты **netstat**. Прямоугольником выделена строка, соответствующая сокету клиента. Сокет находится в состоянии «соединение установлено». На стороне сервера порт клиента отображается правильно.

Самостоятельно

1. провести эксперимент с множеством клиентов при сетевом обмене, представить результаты для виртуальной и реальной сетей;
2. проанализировать пример применения сокетов (сетевой обмен «мгновенными» сообщениями).

Представить архитектуру приложения, алгоритмы сервера и клиента, схему и диаграмму их взаимодействия.

Составить спецификацию функций (имя, назначение, параметры, файлы)

Дополнить приложение, предоставив возможность обмениваться информацией клиентам в Linux и Windows. Исходный код в приложении.

Настроить функционирование в лабораторной сети. Описать необходимые настройки.

3. привести примеры использования портов завершения. Привести пример приложения с большим количеством клиентов до 1000 (когда порты завершения оправданы), общее количество потоков не более 10.
4. оформить приложение с сокетами в виде службы.
5. реализовать обмен на основе UDP

4. Сигналы в Windows

Данное средство IPC в Windows не поддерживается. Однако, например, консольному приложению можно посылать сигналы CTRL+C и CTRL+BREAK. Система может посылать приложению сигналы: CTRL_CLOSE_EVENT, CTRL_LOGOFF_EVENT и CTRL_SHUTDOWN_EVENT, когда пользователь закрывает консоль, выходит из системы, или когда система завершается. По получению данных сигналов процесс может произвести корректное завершение.

С помощью функции *SetConsoleCtrlHandler* можно установить обработчик на данные сигналы, но отправить сигнал другому приложению мы не можем.

Пример. Задание обработчика сигналов завершения для консольного приложения

Установка обработчика сигнала производится с помощью функции:

```
BOOL WINAPI SetConsoleCtrlHandler(PHANDLER_ROUTINE HandlerRoutine, BOOL Add);
```

PHANDLER_ROUTINE HandlerRoutine – функция-обработчик сигнала;
BOOL Add – флаг, значение TRUE которого говорит о добавлении обработчика в список, FALSE – удаление.

Возвращаемое значение: в случае успешной установки обработчика функция возвращает ненулевое значение, иначе – 0.

Функция-обработчик имеет следующее определение:

```
BOOL WINAPI HandlerRoutine(DWORD dwCtrlType);
```

DWORD dwCtrlType – тип сигнала, который был передан функции. Может принимать значение:

- CTRL_C_EVENT - сигнал CTRL+C;
- CTRL_BREAK_EVENT – сигнал CTRL+BREAK;
- CTRL_CLOSE_EVENT – закрытие окна консоли (через меню либо через диспетчер задач);
- CTRL_LOGOFF_EVENT – пользователь выходит из системы;
- CTRL_SHUTDOWN_EVENT – завершение работы системы.

Возвращаемое значение: если обработчик успешно обработал сигнал, он возвращает TRUE, иначе – FALSE.

Зарегистрированный обработчик должен проверять тип сигнала на возможность его обработки.

Обработчики сигналов объединены в список. Когда приходит сигнал, вызывается последний зарегистрированный обработчик (при этом запускается отдельный поток). Если этот обработчик возвращает FALSE (он не обрабатывает этот сигнал), то вызывается следующий. Если все обработчики

вернули FALSE, вызовется обработчик по-умолчанию, который по-умолчанию завершает процесс.

В качестве примера рассмотрим код из msdn. В нем происходит перехват сигналов CTRL+C, CTRL+BREAK. При этом обработчик смотрит, какой сигнал ему передан, и выводит его название. В качестве звуковой индикации работы приложение вызывает функцию Beep. Данная функция воспроизводит звуковой сигнал через динамик консоли с разной частотой и длительностью, задаваемыми ей через параметры.

В функции main регистрируется обработчик сигналов, затем главный поток работает в бесконечном цикле.

Исходный код:

```
#include <windows.h>
#include <stdio.h>
BOOL CtrlHandler( DWORD fdwCtrlType )
{
    switch( fdwCtrlType ) //тип сигнала
    {
        // Handle the CTRL-C signal.
        case CTRL_C_EVENT:
            printf( "Ctrl-C event\n\n" );
            Beep( 750, 300 );
            return( TRUE );

        // CTRL-CLOSE: confirm that the user wants to exit.
        case CTRL_CLOSE_EVENT:
            Beep( 600, 200 );
            printf( "Ctrl-Close event\n\n" );
            return( TRUE );

        // Pass other signals to the next handler.
        case CTRL_BREAK_EVENT:
            Beep( 900, 200 );
            printf( "Ctrl-Break event\n\n" );
            return FALSE;

        case CTRL_LOGOFF_EVENT:
            Beep( 1000, 200 );
            printf( "Ctrl-Logoff event\n\n" );
            return FALSE;

        case CTRL_SHUTDOWN_EVENT:
            Beep( 750, 500 );
            printf( "Ctrl-Shutdown event\n\n" );
            return FALSE;

        default:
```

```

        return FALSE;
    }
}

int main( void )
{
    if( SetConsoleCtrlHandler( (PHANDLER_ROUTINE) CtrlHandler, TRUE ) )
    {
        printf( "\nThe Control Handler is installed.\n" );
        printf( "\n -- Now try pressing Ctrl+C or Ctrl+Break, or" );
        printf( "\n    try logging off or closing the console...\n" );
        printf( "\n(...waiting in a loop for events...)\n\n" );

        while( 1 ){ }
    }
    else
    {
        printf( "\nERROR: Could not set control handler");
        return 1;
    }
    return 0;
}

```

Результаты работы программы представлены на рис. 27.

```

C:\windows\system32\cmd.exe
Ctrl-Break event
^C
D:\учеба\OS\Отчеты\Лаба 7\Доп обработчик сигналов завершения\ConsoleObrab\Release>ConsoleObrab.exe
The Control Handler is installed.
 -- Now try pressing Ctrl+C or Ctrl+Break, or
    try logging off or closing the console...
<...waiting in a loop for events...>
Ctrl-C event
Ctrl-C event
Ctrl-Break event
^C
D:\учеба\OS\Отчеты\Лаба 7\Доп обработчик сигналов завершения\ConsoleObrab\Release>

```

Рис. 27. Результат работы программы обработчика сигналов завершения

Для сигналов воспроизводились разные по тональности звуки.

Самостоятельно предложить собственную реализацию обработчика сигнала.

5. Разделяемая память

Потоки одного процесса могут разделять общую память этого процесса. У каждого процесса – свое изолированное адресное пространство. Кроме рассмотренных выше средств передачи информации между процессами или потоками разных процессов, одно из наиболее эффективных – использование общей памяти, доступ к которой обеспечивается со стороны каждого процесса. ОС Windows поддерживает такое средство, как **именованная, совместно используемая память**.

Приведем **системные функции**, которые позволяют запрограммировать такое взаимодействие.

Первый участвующий в обмене информацией процесс создает объект "проекция файла" при помощи вызова функции *CreateFileMapping()* с параметром *INVALID_HANDLE_VALUE* и именем для объекта. Используя флажок *PAGE_READWRITE*, задается доступ по чтению и записи в память через представление данных файла в адресном пространстве процесса. Процесс затем использует дескриптор объекта "проекция файла", возвращаемый функцией *CreateFileMapping()*, при вызове функции *MapViewOfFile()*. Эта функция создает представление файла в адресном пространстве процесса и возвращает указатель на представление данных файла для их дальнейшего использования.

Другой процесс может получить доступ к тем же данным при помощи вызова функции *OpenFileMapping()* с тем же самым именем, что и первый процесс, а затем использовать функцию *MapViewOfFile()*, чтобы получить свой указатель на представление данных файла.

Для записи данных в память используется функция *CopyMemory()*, первый аргумент которой – возвращаемый функцией *MapViewOfFile()* указатель, а следующие – характеризуют записываемые данные.

Когда процессу больше не нужен доступ к объекту "проекция файла в память", он должен вызвать функцию *CloseHandle()* для дальнейшего освобождения ресурса. При условии, что все дескрипторы закрыты (не осталось процессов, использующих этот ресурс), система может освободить секцию файла подкачки, используемого объектом.

Пример. Взаимодействие двух процессов через совместно используемую именованную память, при котором первый процесс записывает данные, а второй считывает их.

Задание. Создать программу, в которой первый процесс генерирует случайное число и записывает его в буфер, доступный второму процессу, откуда он его и считывает с последующим выводом.

Исходный код первого процесса:

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#define BUF_SIZE 256
TCHAR szName[]=TEXT("MyFileMappingObject");
TCHAR szMsg[]=TEXT("Message from first process");
HANDLE WINAPI mutex;
void main()
{
    HANDLE hMapFile;
    LPCTSTR pBuf;
    mutex = CreateMutex(NULL, false, TEXT("SyncMutex"));
    // create a memory, with two process will be working
    hMapFile = CreateFileMapping(
        INVALID_HANDLE_VALUE, // использование файла подкачки
        NULL, // защита по умолчанию
        PAGE_READWRITE, // доступ к чтению/записи
        0, // макс. размер объекта
        BUF_SIZE, // размер буфера
        szName); // имя отраженного в памяти объекта

    if (hMapFile == NULL || hMapFile == INVALID_HANDLE_VALUE)
    {
        printf("Не может создать отраженный в памяти объект (%d).\n",
            GetLastError());
        return;
    }
    pBuf = (LPTSTR)MapViewOfFile(hMapFile, // дескриптор проецируемого
    // в памяти объекта
        FILE_MAP_ALL_ACCESS, // разрешение
    // чтения/записи (режим доступа)
        0, // Старшее слово смещения файла, где
    // начинается отображение
        0, // Младшее слово смещения файла, где
    // начинается отображение
        BUF_SIZE); // Число отображаемых байтов файла

    if (pBuf == NULL)
    {
        printf("Представление проецированного файла невозможно (%d).\n",
            GetLastError());
        return;
    }
    int i=0;
    while(true)
    {
```

```

        i=rand();
        itoa(i, (char *)szMsg, 10);
        WaitForSingleObject(mutex, INFINITE);
            CopyMemory((PVOID)pBuf, szMsg, sizeof(szMsg));
        printf("write message: %s\n", (char *)pBuf);
//Sleep(1000); //необходимо только для отладки - для удобства
//представления и анализа результатов
        ReleaseMutex(mutex);
    }
    // освобождение памяти и закрытие описателя handle
    UnmapViewOfFile(pBuf);
    CloseHandle(hMapFile);
    CloseHandle(mutex);
}

```

Для организации синхронизации доступа к памяти в данном примере рассматривается использование мьютексов (более подробная информация о средствах синхронизации изложена в следующем разделе). В первом процессе создается **именованный мьютекс**, который "защищает" **критические участки кода**, в данном случае - *запись в общую разделяемую память*.

Записываемая в буфер информация - это случайное число, которое генерируется функцией rand() перед каждой записью.

Второй процесс открывает ранее созданный и именованный первым процессом мьютекс.

Исходный код второго процесса:

```

#include <windows.h>
#include <stdio.h>
#include <conio.h>
#define BUF_SIZE 256
#define TIME 15 // number of reading operation in this process
TCHAR szName[]=TEXT("MyFileMappingObject");
HANDLE WINAPI mutex;
void main()
{
    HANDLE hMapFile;
    LPCTSTR pBuf;
    mutex = OpenMutex(
        MUTEX_ALL_ACCESS, // request full access
        FALSE, // handle not inheritable
        TEXT("SyncMutex")); // object name
    if (mutex == NULL)
        printf("OpenMutex error: %d\n", GetLastError() );
    else printf("OpenMutex successfully opened the mutex.\n");
    hMapFile = OpenFileMapping(
        FILE_MAP_ALL_ACCESS, // доступ к чтению/записи
        FALSE, // имя не наследуется
        szName); // имя "проецируемого" объекта
    if (hMapFile == NULL)
    {

```

```

        printf("Невозможно открыть объект проекция файла (%d).\n",
GetLastError());
        return;
    }
    pBuf =(LPTSTR) MapViewOfFile(hMapFile,        // дескриптор
//"проецируемого" объекта
        FILE_MAP_ALL_ACCESS, // разрешение чтения/записи
        0,
        0,
        BUF_SIZE);
    if (pBuf == NULL)
    {
        printf("Представление проецированного файла (%d) невозможно .\n",
GetLastError());
        return;
    }
    for(int i=0; i<TIME; i++)
    {
        WaitForSingleObject(mutex, INFINITE);
        printf("read message: %s\n", (char *)pBuf);
        ReleaseMutex(mutex);
    }
    UnmapViewOfFile(pBuf);
    CloseHandle(hMapFile);
}

```

Первый процесс завершается по Ctrl+C, а второй считывает данные фиксированное количество раз (задаваемое переменной TIME).

Пример выполнения:

```

write message: 18467
write message: 6334
write message: 26500
write message: 19169
write message: 15724
write message: 11478
write message: 29358
write message: 26962
write message: 24464
write message: 5705
write message: 28145
write message: 23281
write message: 16827
write message: 9961
write message: 491
write message: 2995
write message: 11942
^CДля продолжения нажмите любую клавишу . . .

C:\WINDOWS\system32\cmd.exe
OpenMutex successfully opened the mutex.
read message: 18467
read message: 6334
read message: 26500
read message: 19169
read message: 15724
read message: 11478
read message: 29358
read message: 26962
read message: 24464
read message: 5705
read message: 28145
read message: 23281
read message: 16827
read message: 9961
read message: 491
Для продолжения нажмите любую клавишу

```

Рис.28. Пример работы программ с использованием разделяемой памяти

Каждый процесс ассоциирован со своим терминалом и выводит информацию в свое окно.

6. Почтовые слоты (MailSlot)

MailSlot – механизм синхронизации, иначе называемый «почтовый ящик». Каждый слот реализуется как псевдофайл в оперативной памяти и содержит некоторое количество записей («сообщений»), которые могут быть прочтены всеми компьютерами в сетевом домене. Общий размер данных не должен превышать 64К. В отличие от дисковых файлов, файлы MailSlot временные. Когда все указатели на MailSlot закрываются, MailSlot и все данные, которые он содержит, удаляются.

Для обмена посредством MailSlot создается клиент-серверное приложение. MailSlot сервер – является процессом, который создает и владеет MailSlot. При создании сервер получает указатель, используемый затем при чтении или записи данных им самим или другим процессом, получившим указатель на MailSlot. Создать слот можно только локально, а получать доступ (обращаться) и локально и удалённо. Для создания почтовых слотов используются имена в следующей форме: [\\.\mailslot\\[path\]name](#) . Форма имени при открытии почтового слота: [\\ComputerName\mailslot\\[path\]name](#) или [\\DomainName\mailslot\\[path\]name](#)

Для доступа по записи используют **стандартные файловые операции получения дескриптора и работы с ним** (*CreateFile()*, *WriteFile()*, *CloseHandle()*).

При создании почтовых слотов *с одинаковым именем на нескольких компьютерах* домена возможна **широковещательная рассылка** сообщений клиентов. Осуществление записи от множества клиентов может осуществляться одновременно. Сообщения располагаются в почтовом слоте в очереди в порядке записи до тех пор, пока поочередно (по одному за раз) не будут считаны (в той же очередности).

Для создания почтового слота используется функция:

HANDLE CreateMailslot

```
{
    LPCTSTR lpName,                // имя
    DWORD nMaxMessageSize,        // максимальный размер
    DWORD lReadTimeout,           // интервал тайм-аута
    чтения
    LPSECURITY_ATTRIBUTES lpSecurityAttributes // информация о
};                                //безопасности
```

При успешном выполнении возвращается указатель, а в случае неудачи - INVALID_HANDLE_VALUE.

Для получения информации о наличии данных MailSlot, используется функция:

BOOL GetMailslotInfo

```
(  
    HANDLE hMailslot,           // указатель на слот  
    LPDWORD lpMaxMessageSize, // максимальный размер  
    LPDWORD lpNextSize,       // размер следующего  
    LPDWORD lpMessageCount,   // количество сообщений  
    LPDWORD lpReadTimeout     // тайм-аут.  
);
```

Возвращаемое значение равно 0, если слот не существует или произошла ошибка.

Самостоятельно

- предложить собственную реализацию приложения, иллюстрирующую обмен информацией почтовыми слотами,
- продемонстрировать возможность локального и удаленного доступа,
- выполнить широковещательную передачу данных.

Раздел 4. Средства синхронизации потоков и процессов в ОС Windows и их применение

Целью синхронизации доступа к ресурсам потоков и процессов является обеспечение принципа взаимного исключения для сохранения целостности ресурса и предотвращения взаимных блокировок.

Синхронизация потоков и процессов в семействе ОС Windows осуществляется, как правило, сочетанием различных средств, к которым относятся: объекты синхронизации и функции ожидания завершения и блокировок. К объектам синхронизации относят:

- мьютексы;
- семафоры;
- события;
- критические секции (CRITICAL_SECTION (CS));

условные переменные (вспомогательное средство синхронизации, используемое в сочетании с другими объектами);

таймеры ожидания;

порты завершения (ввода/вывода);

Первые три типа объектов — мьютексы, семафоры и события — являются объектами ядра, имеющими дескрипторы. Как следствие, они могут использоваться как потоками одного процесса, так и потоками разных процессов, т.е. являются универсальными средствами ИРС для процессов и потоков.

Объект-мьютекс позволяет синхронизировать доступ к ресурсу сразу нескольких потоков из разных процессов; при этом можно задать максимальное время ожидания доступа к ресурсу. Одновременно доступ к мьютексу имеет только 1 поток.

Объекты ядра «семафор» используются для учета ресурсов. Но, кроме того, что, как и все объекты ядра, они содержат счетчик числа пользователей, семафоры имеют счетчики ресурсов: один определяет максимальное число ресурсов (контролируемое семафором), другой используется как счетчик текущего числа ресурсов. Семафор позволяет захватить себя несколькими потоками, после чего захват будет невозможен, пока один из ранее захвативших семафор потоков не освободит его. Семафоры применяются для ограничения количества потоков, одновременно работающих с ресурсом (или ресурсами).

Семафоры и мьютексы могут иметь имя, через которое другие процессы могут получить доступ к объектам. Разница состоит в том, что при межпроцессной синхронизации в каждом процессе должен быть получен свой дескриптор к одному и тому же объекту ядра.

События — самая примитивная разновидность объектов ядра. Они содержат счетчик числа пользователей (как и все объекты ядра) и две булевы переменные: одна сообщает тип данного объекта-события, другая — его состояние (свободен или занят). События просто уведомляют об окончании какой-либо операции и бывают двух типов: со сбросом вручную (manual-reset events) и с автосбросом (auto-reset events). Первые позволяют возобновлять выполнение сразу нескольких ждущих потоков, вторые — только одного. События используются также для других целей, например, для асинхронного ввода/вывода.

Критические секции (CS) являются объектами синхронизации прикладного режима, не имеют дескрипторов и не могут совместно использоваться разными процессами. Критическая секция помогает выделить

участок кода, где поток получает доступ к разделяемому ресурсу, и предотвратить одновременное использование ресурса. К критической секции в одно время имеет доступ только 1 поток. Данное средство не может быть применено для синхронизации процессов, только для синхронизации потоков одного процесса. Поскольку критическая секция не является объектом ядра, то и использовать функции ожидания (Wait...) не представляется возможным. Соответствующие переменные должны объявляться как переменные типа CRITICAL_SECTION, их можно инициализировать и удалять. В соответствии с принципом взаимного исключения потоки входят в объекты CS и покидают их, но выполнение кода отдельного объекта CS каждый раз разрешено только одному потоку. Вместе с тем, один и тот же поток может входить в несколько отдельных объектов CS и покидать их, если они расположены в разных местах программы.

Условные переменные являются вспомогательным средством, применяются, как правило, в сочетании с каким-либо основным средством синхронизации (например, с критическими секциями) и используются как альтернативная возможность методу периодического опроса состояния доступа к ресурсу.

Алгоритмически в случае использования только одного объекта синхронизации (кроме критических секций) приходится использовать ожидающие функции, которые возвращают управление, если все или часть переданных им объектов свободны. Т.е. они захватывают объект-синхронизации, проверяют, могут ли использовать ресурс. Если не могут, то просто освобождают его и опять становятся в очередь к объекту. Таким образом, в алгоритме осуществляется периодический опрос возможности доступа к ресурсу.

Сочетаемость и увеличение количества объектов синхронизации позволяет избавиться от необходимости периодического опроса и пробуждать потоки тогда, когда это действительно необходимо, т.е. когда доступ к ресурсу открыт.

Рассмотрим примеры, позволяющие продемонстрировать применение тех или иных средств синхронизации, и попытаемся проанализировать какое средство или сочетание средств и в каких ситуациях предпочтительнее.

Пример 1. Создание двух потоков: потока – производителя и потока – потребителя.

Задание. Поток разделяет целочисленный массив, в который заносятся производимые и извлекаются потребляемые данные. Для наглядности и контроля за происходящим в буфер помещается наращиваемое значение, однозначно идентифицирующее производителя и номер его очередной посылки.

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.).

Задание необходимо выполнить **различными способами**, применив следующие средства синхронизации доступа к разделяемому ресурсу:

Мьютексы; Семафоры; Критические секции; Объекты события; Условные переменные; Функции ожидания.
--

Создать аналогичные программы для множества потоков, количество которых можно задать из командной строки.

Программы должны предоставлять возможность завершения по таймеру либо по команде оператора

Пример выполнения

Для демонстрации применения перечисленных средств синхронизации целесообразно создать некий шаблон программы, в которой поочередно использовать эти средства для разделения одного и того же ресурса в одинаковых условиях. Это позволит сравнить результаты между собой и увидеть достоинства и недостатки каждого из средств для конкретной ситуации. Анализируя каждый пример применения, можно обозначить область предпочтительного использования того или иного средства синхронизации.

Создание программы-шаблона

Главный поток определяет конфигурацию программы, создает все структуры данных и потоки. После создания всех объектов поток ожидает завершения созданных потоков, а затем производит удаление всех созданных объектов.

Поток-писатель – размещает сообщения в очереди.

Поток-читатель – выводит на экран сообщения из очереди.

В программе предусматривается возможность настройки:

- скоростей читателя и писателя (путем установки задержек на работу, одной – для всех читателей, одной – для всех писателей);
- времени жизни приложения (либо по таймеру, либо по запросу оператора);
- размера очереди сообщений;
- количества писателей и читателей;

Данные параметры загружаются в структуру вида:

```
struct Configuration
{
    int numOfReaders; //число потоков-читателей
    int numOfWriters; //число потоков-писателей
    int sizeofQueue; //размер очереди
    int readersDelay; //задержка на работу читателей (в
миллисек)
    int writersDelay; //задержка на работу писателей (в
миллисек)
    int ttl; // "время жизни"
};
```

ttl – время жизни приложения. Если **ttl**>0, то завершение приложения происходит через **ttl** секунд (временной интервал измеряется с помощью Waitable таймера). Если **ttl**<0, то завершение работы происходит по запросу оператора.

Объект данной структуры является глобальной переменной.

Вся конфигурационная информация загружается из файла, имеющего следующий вид:

```
NumOfReaders= 10
ReadersDelay= 100
NumOfWriters= 10
WritersDelay= 200
SizeOfQueue= 10
ttl= 3
```

Пусть в качестве общего ресурса используется очередь **FIFO** (First Input First Output). Потоки-писатели добавляют в очередь сообщения, потоки-читатели – забирают.

Очередь определяется в виде структуры:

```
struct FIFOQueue //FIFO очередь
{
    char **data; //массив сообщений
    int writeindex; //индекс записи
    int getindex; //индекс чтения
};
```

```

int size; //размер очереди
short full; //очередь заполнена
};

```

Схема алгоритма работы главного потока представлен на рис. 29.



Рисунок 19. Схема алгоритма работы главного потока

Исходный код функции главного потока:

```

//глобальные переменные
struct FIFOQueue queue; //структура очереди
struct Configuration config; //конфигурация программы
bool isDone=false;
HANDLE *allhandlers; //массив всех создаваемых потоков
int main(int argc, char* argv[])
{
    char filename[30]; //имя файла конфигурации
    if(argc<2)
    {
        //используем конфигурацию по-умолчанию
        strcpy(filename,DEF_CONFIGFILE);
    }
    else
    {
        strcpy(filename,argv[1]);
    }
    //функция установки конфигурации (передаем имя читаемого файла и заполняемую
    //структуру)
    SetConfig(filename,&config);
}

```

```

//создаем необходимые потоки без их запуска
CreateAllThreads(&config);
//Инициализируем очередь
queue.full=0;
queue.getindex=0;
queue.writeindex=0;
queue.size=config.sizeOfQueue;
queue.data=new char*[config.sizeOfQueue];
//инициализируем средство синхронизации,
//здесь размещаем код применения выбранного средства синхронизации
. . .

//запускаем потоки на исполнение
for(int i=0;i<config.numOfReaders+config.numOfWriters+1;i++)
ResumeThread(allhandlers[i]);

//ожидаем завершения всех потоков
WaitForMultipleObjects(config.numOfReaders+config.numOfWriters+1,
allhandlers,TRUE,INFINITE);

for(int i=0;i<config.numOfReaders+config.numOfWriters+1;i++)
//закрываем handle потоков
CloseHandle(allhandlers[i]);
//удаляем объект синхронизации

printf("all is done\n");
return 0;
}

```

Для структурирования кода использованы вспомогательные функции.

Функция **SetConfig** – производит чтение конфигурационного файла и заполнение структуры config.

Функция **CreateAllThreads** – создает все потоки (читатели, писатели, координатор) с параметром **CREATE_SUSPENDED** (после создания потоки не начнут работу, пока не будет вызвана функция ResumeThread).

Код функций: **SetConfig** , **CreateAllThreads** представлен в приложении ниже.

Функция **WaitForMultipleObjects** – возвращает управление в случае, когда «освободились» все (или 1 из параметров) HANDLE из переданного ей массива. Поведение её зависит от третьего параметра. Если он равен TRUE, то управление вернется либо когда освободятся все HANDLE, либо когда истечет таймаут (последний параметр), если он равен FALSE, то управление вернется при освобождении хотя бы 1 объекта или по таймауту.

```

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

```

Приложение

Исходный код вспомогательных функций:

```
HANDLE CreateAndStartWaitableTimer(int sec) //создание, установка
//и запуск таймера
{
    __int64 end_time;
    LARGE_INTEGER end_time2;
    HANDLE tm = CreateWaitableTimer(NULL, false, "timer");
    end_time = -1 * sec * 10000000;
    end_time2.LowPart = (DWORD) (end_time & 0xFFFFFFFF);
    end_time2.HighPart = (LONG) (end_time >> 32);
    SetWaitableTimer(tm, &end_time2, 0, NULL, NULL, false);
    return tm;
}

void CreateAllThreads(struct Configuration * config) //создание
//всех потоков
{
    printf("createConfig:\n NumOfreadrs = %d | ReadersDelay= %d |
    NumOfwriters= %d | WritersDelay = %d | sizeofqueue = %d | ttl =
    %d\n", config->numOfReaders, config->readersDelay, config-
    >numOfWriters, config->writersDelay, config->sizeOfQueue, config-
    >ttl);
    allhandlers=new HANDLE[config->numOfReaders+config->
    numOfWriters+1];
    int count=0;

    printf("create readers\n");
    for(int i=0;i<=config->numOfReaders-1;i++, count++) //создаем
    потоки-читатели
    {
        printf("count= %d\n",count);
        if((allhandlers[count]=CreateThread(NULL,0,ThreadReaderHandler,
        (LPVOID)i,CREATE_SUSPENDED,NULL))!=NULL) //создаем потоки-
        //читатели, которые пока не стартуют
        {
            printf("impossible to create thread-reader\n
            %d",GetLastError());
            exit(8000);
        }
    }
    printf("create writers\n");
    for(int i=0;i<=config->numOfReaders-1;i++, count++) //создаем
    //потоки-писатели
    {
        printf("count= %d\n",count);

        if((allhandlers[count]=CreateThread(NULL,0,ThreadWriterHandler
        ,(LPVOID)i,CREATE_SUSPENDED,NULL))!=NULL) //создаем потоки-
        //читатели, которые пока не стартуют
```

```

        {
            printf("impossible to create thread-writer\n");
            exit(8001);
        }
    }
    //создаем поток TimeManager
    printf("create TimeManager\n");
    printf("count= %d\n",count);
    if((allhandlers[count]=CreateThread(NULL,0,ThreadTimeManagerHandler,
ndler,(LPVOID)config->ttd,CREATE_SUSPENDED,NULL))!=NULL)
    //создаем потоки-читатели, которые пока не стартуют
    {
        printf("impossible to create thread-reader\n");
        exit(8002);
    }
    printf("successfully created threads\n");
    return;
}

void SetConfig(char * filename,struct Configuration * config)
//функция установки конфигурации
{
    FILE *f;
    int numOfReaders;
    int numOfWriters;
    int readersDelay;
    int writersDelay;
    int sizeOfQueue;
    int ttl;
    char tmp[20];
    if((f=fopen(filename,"r"))!=NULL)
    {
        printf("impossible open config file %s\n",filename);
        exit(1000);
    }
    //Вид конфигурационного файла
    //NumOfReaders= 10
    //ReadersDelay= 100
    //NumOfWriters= 10
    //WritersDelay= 200
    //SizeOfQueue= 10
    //ttl= 3
    //начинаем читать конфигурацию
    fscanf(f,"%s %d",tmp,&numOfReaders); //число потоков-читателей

    fscanf(f,"%s %d",tmp,&readersDelay); //задержки потоков-
читателей
    fscanf(f,"%s %d",tmp,&numOfWriters); //число потоков-писателей
    fscanf(f,"%s %d",tmp,&writersDelay); //задержки потоков-
писателей
    fscanf(f,"%s %d",tmp,&sizeOfQueue); //размер очереди
    fscanf(f,"%s %d",tmp,&ttl); //время жизни
}

```



```

if(numOfReaders<=0 || numOfWriters<=0)
{
    printf("incorrect num of Readers or writers\n");
    exit(500);
}
else if(readersDelay<=0 || writersDelay<=0)
{
    printf("incorrect delay of Readers or writers\n");
    exit(501);
}
else if(sizeOfQueue<=0)
{
    printf("incorrect size of queue\n");
    exit(502);
}
else if(ttl==0)
{
    printf("incorrect ttl\n");
    exit(503);
}
config->numOfReaders=numOfReaders;
config->readersDelay=readersDelay;
config->numOfWriters=numOfWriters;
config->writersDelay=writersDelay;
config->sizeOfQueue=sizeOfQueue;
config->ttl=ttl;
return;
}

```

Код вспомогательных функций лучше разместить в отдельном файле, например, utils.cpp.

Схема алгоритма работы потока-координатора представлена на рис. 30.

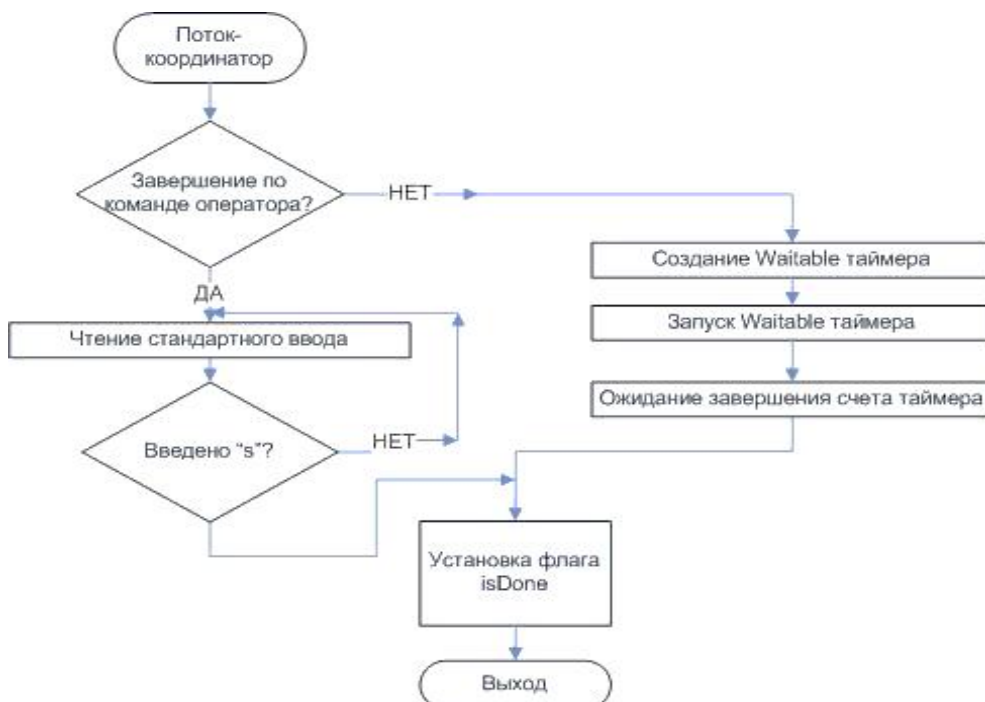


Рисунок 30. Схема алгоритма работы потока-координатора

Схема алгоритма работы потока-читателя представлена на рис. 31.

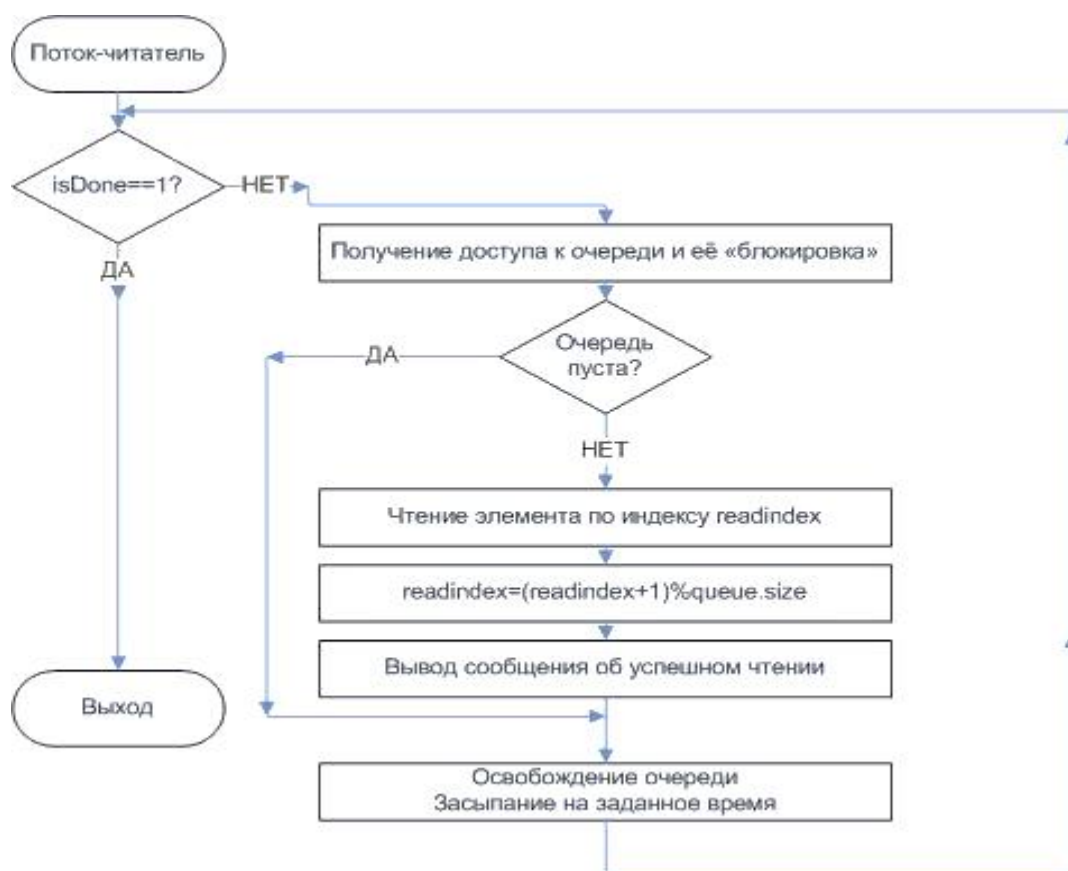


Рисунок 31. Схема алгоритма работы потока-читателя

Исходный код функции потока-координатора:

```
DWORD WINAPI ThreadTimeManagerHandler(LPVOID prm)
{
    int ttl=(int) prm;
    if(ttl<0)
    {
        char buf[100]; //завершение по команде оператора
        while(1)
        {
            fgets(buf,sizeof(buf),stdin);
            if(buf[0]=='s');
            {
                isDone=true;
                break;
            }
        }
    }
    else
    {
        HANDLE h = CreateAndStartWaitableTimer(ttl); //завершение по таймеру
```

```

    WaitForSingleObject(h, INFINITE);
    isDone = true;
    CloseHandle(h);
}
printf("TimeManager finishing work\n");
return 0;
}

```

Функция **CreateAndStartWaitableTimer** – создает и запускает Waitable таймер. Код функции представлен в приложении выше.

Исходный код потока-читателя:

```

DWORD WINAPI ThreadReaderHandler(LPVOID prm)
{
    int myid=(int)prm;

    while(isDone!=true)
    {
        //Захват объекта синхронизации, здесь размещается код
//применения выбранного средства
        . . .

        if(queue.readindex!=queue.writeindex || queue.full==1) //если
//в очереди есть данные
        {
            queue.full=0; //взяли данные, значит очередь не пуста

            printf("Reader %d get data: \"%s\" from position
%d\n",myid,queue.data[queue.readindex],queue.readindex);
//печатаем принятые данные
            free(queue.data[queue.readindex]); //очищаем очередь от
данных
            queue.data[queue.readindex]=NULL;

            queue.readindex=(queue.readindex+1)%queue.size;
        }
        //Освобождение объекта синхронизации, здесь размещаем код
//применения выбранного средства
        . . .

        Sleep(config.readersDelay); //задержка
    }
    printf("Reader %d finishing work\n",myid);
    return 0;
}

```

Схема алгоритма работы потоков-писателей представлена на рис. 32.

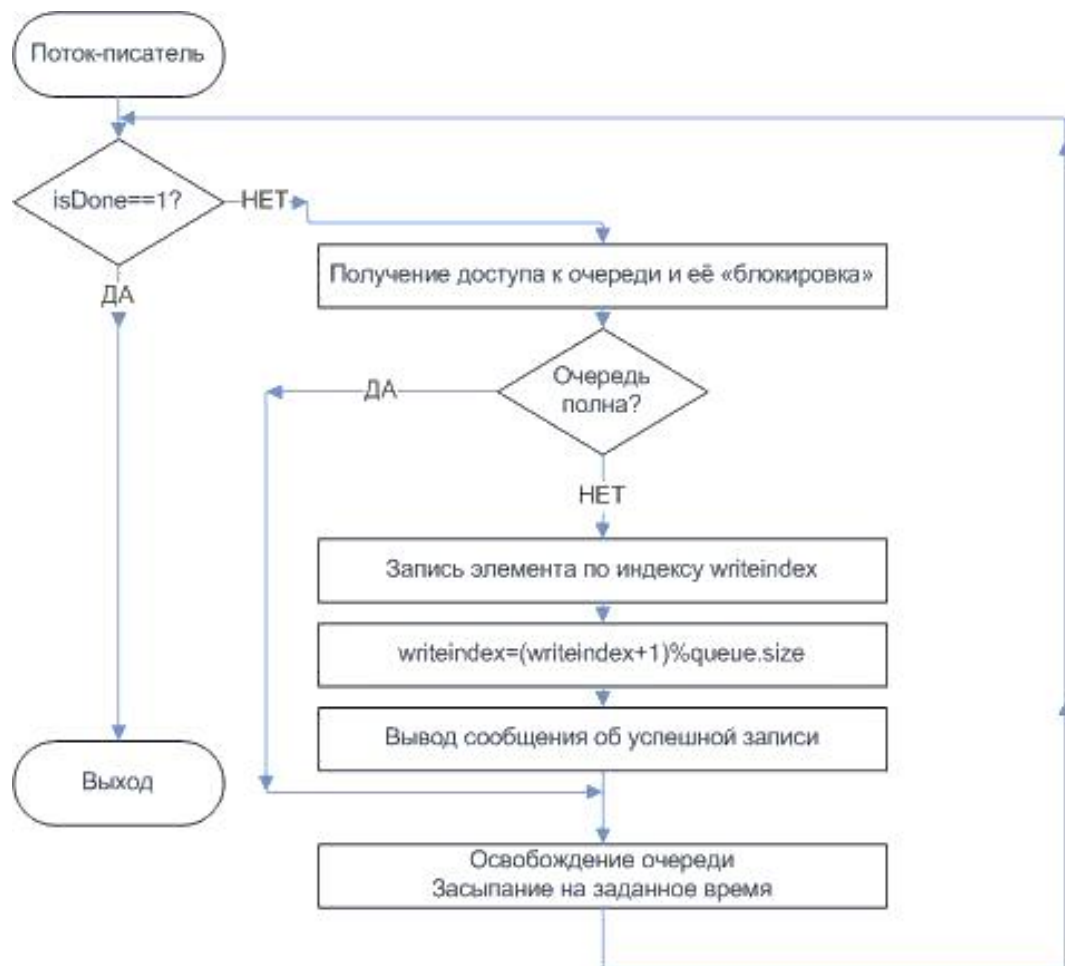


Рис. 32. Схема алгоритма работы потока-писателя

Исходный код потока-писателя:

```

DWORD WINAPI ThreadWriterHandler(LPVOID prm)
{
    int myid=(int)prm;
    char tmp[50];
    int msgnum=0; //номер передаваемого сообщения
    while(isDone!=true)
    {
        //Захват синхронизирующего объекта, здесь размещаем код
        //применения выбранного средства
        . . .

        if(queue.readindex!=queue.writeindex || !queue.full==1)
        //если в очереди есть место
        {

            //заносим в очередь данные
            sprintf(tmp,"writer_id = %d numMsg= %3d",myid,msgnum);
            queue.data[queue.writeindex]=strdup(tmp);
            msgnum++;
            printf("Writer %d put data: \"%s\" in position
            %d\n",myid,queue.data[queue.writeindex],queue.writeindex);
            //печатаем принятые данные
  
```

```

        queue.writeindex=(queue.writeindex+1)%queue.size;
        queue.full=queue.writeindex==queue.readindex ? 1 : 0;
//если очередь заполнилась
    }

    //освобождение объекта синхронизации, здесь размещаем код
применения выбранного средства
        . . .

    Sleep(config.writersDelay); //задержка

}
printf("Writer %d finishing work\n",myid);
return 0;
}

```

В дальнейшем алгоритм работы потоков читателя, писателя, координатора изменяться не будет, а меняются только применяемые средства синхронизации и соответствующий код. Место размещения этого кода обозначено в комментариях жирным шрифтом.

4.1. Использование мьютексов в качестве средства синхронизации

Сведения об используемом средстве синхронизации

Объекты ядра «мьютексы» гарантируют потокам взаимоисключающий доступ к единственному ресурсу. Отсюда и произошло название этих объектов (mutual exclusion, mutex). Они содержат счетчик числа пользователей, счетчик рекурсии и переменную, в которой запоминается идентификатор потока.

Мьютексы ведут себя точно так же, как и критические секции (см. соответствующий пункт ниже). Однако, если последние являются объектами пользовательского режима, то мьютексы — объектами ядра. Кроме того, единственный объект-мьютекс позволяет синхронизировать доступ к ресурсу нескольких потоков из разных процессов; при этом можно задать максимальное время ожидания доступа к ресурсу.

Идентификатор потока определяет, какой поток захватил мьютекс, а счетчик рекурсий — сколько раз. У мьютексов много применений, и это наиболее часто используемые объекты ядра. Как правило, с их помощью защищают блок памяти, к которому обращается множество потоков. Если бы потоки одновременно использовали какой-то блок памяти, то данные в нем были бы повреждены. Мьютексы гарантируют, что любой поток получает

монопольный доступ к блоку памяти, и тем самым обеспечивают целостность данных.

Для мьютексов определены следующие правила:

- если его идентификатор потока равен 0 (у самого потока не может быть такой идентификатор), мьютекс не захвачен ни одним из потоков и находится в свободном состоянии;
- если его идентификатор потока не равен 0, мьютекс захвачен одним из потоков и находится в занятом состоянии;
- если идентификатор вызывающего потока равен идентификатору в мьютексе, то счетчик рекурсии увеличивается на 1 и поток получает доступ.

Для создания мьютекса используется функция:

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

PSECURITY_ATTRIBUTES psa – параметры безопасности;

BOOL fIniLialOwner - начальное состояние мьютекса. Если параметр равен False, то создаваемый мьютекс никому изначально не принадлежит. Если параметр равен TRUE, то мьютекс сразу захватывается вызывающим функцию CreateMutex потоком;

PCTSTR pszName – имя мьютекса.

Для «открытия мьютекса» используется функция:

```
HANDLE OpenMutex( DWORD fdwAccess, 800L bInheritHandle,  
PCTSTR pszName );
```

DWORD fdwAccess – права доступа к мьютексу. Могут быть указано только SYNCHRONIZE;

800L bInheritHandle – флаг, определяющий, будет ли описатель созданного мьютекса наследоваться;

PCTSTR pszName – имя мьютекса.

Для мьютексов сделано одно исключение в правилах перехода объектов ядра из одного состояния в другое. Допустим, поток ждет освобождения занятого объекта мьютекса. В этом случае поток обычно засыпает (переходит в состояние ожидания). Однако система проверяет, не совпадает ли идентификатор потока, пытающегося захватить мьютекс, с аналогичным идентификатором у мьютекса. Если они совпадают, система по-прежнему

выделяет потоку процессорное время, хотя мьютекс все еще занят. Подобных особенностей в поведении нет ни у каких других объектов ядра в системе. Всякий раз, когда поток захватывает объект-мьютекс, счетчик рекурсии в этом объекте увеличивается на 1. Единственная ситуация, в которой значение счетчика рекурсии может быть больше 1, — поток захватывает один и тот же мьютекс несколько раз, пользуясь упомянутым исключением из общих правил.

Для доступа к ресурсу используются Wait функции: **WaitForSingleObject** и другие.

Мьютекс освобождается вызовом функции:

```
BOOL ReleaseMutex(HANDLE hMutex);
```

Эта функция уменьшает счетчик рекурсии в объекте-мьютексе на 1. Если данный объект передавался во владение потоку неоднократно, поток обязан вызвать **ReleaseMutex** столько раз, сколько необходимо для обнуления счетчика рекурсии. Как только счетчик станет равен 0, переменная, хранящая идентификатор потока, тоже обнулится, и объект-мьютекс освободится. После этого система проверит, ожидают ли освобождения мьютекса какие-нибудь другие потоки. Если да, система «по-честному» выберет один из ждущих потоков и передаст ему во владение объект-мьютекс.

Функция возвращает значение TRUE, если вызов прошел успешно, или FALSE – если нет (мьютекс нам не принадлежал).

Применение мьтексов и изменения в исходном коде шаблона

В список глобальных переменных была добавлена еще одна типа **HANDLE**, хранящая описатель мьютекса:

```
HANDLE mutex;
```

Основной поток производит инициализацию мьютекса и его освобождение:

```
int main(int argc, char* argv[])
{

    char filename[30]; //имя файла конфигурации
    if(argc<2)
    {
        ...
        queue.data=new char*[config.sizeOfQueue];

        //инициализируем средство синхронизации
        mutex=CreateMutex(NULL, FALSE, "" );

        //запускаем потоки на исполнение
```

```

for(int i=0;i<config.numOfReaders+config.numOfWriters+1;i++)
    ResumeThread(allhandlers[i]);

//ожидаем завершения всех потоков
WaitForMultipleObjects(config.numOfReaders+config.numOfWriters
+1,allhandlers,TRUE,INFINITE);
for(int i=0;i<config.numOfReaders+config.numOfWriters+1;i++)
//закрываем handle потоков
    CloseHandle(allhandlers[i]);
//удаляем объект синхронизации
CloseHandle(mutex);

printf("all is done\n");

return 0;
}

```

Потоки читатели и писателя пробуют захватить мьютекс функцией **WaitForSingleObject**. При успешном захвате выполняют операции с очередью, затем освобождают с помощью функции **ReleaseMutex**:

```

DWORD WINAPI ThreadReaderHandler(LPVOID prm)
{
    int myid=(int)prm;

    while(isDone!=true)
    {
        //Захват объекта синхронизации
        WaitForSingleObject(mutex,INFINITE);
        if(queue.readindex!=queue.writeindex || queue.full==1) //если
//в очереди есть данные
        {
            ...
            queue.readindex=(queue.readindex+1)%queue.size;
        }
        //Освобождение объекта синхронизации
        ReleaseMutex(mutex);
        Sleep(config.readersDelay); //задержка
    }
    printf("Reader %d finishing work\n",myid);
    return 0;
}

```

```

DWORD WINAPI ThreadWriterHandler(LPVOID prm)
{
    int myid=(int)prm;
    char tmp[50];
    int msgnum=0;
    while(isDone!=true)
    {
        //Захват синхронизирующего объекта
        WaitForSingleObject(mutex,INFINITE);

```



```

    if(queue.readindex!=queue.writeindex || !queue.full==1)
//если в очереди есть место
    {
        ...
        queue.full=queue.writeindex==queue.readindex ? 1 : 0;
//если очередь заполнилась
    }
    // освобождение синхронизируемого объекта
    ReleaseMutex(mutex);
    Sleep(config.writersDelay); //задержка
}
printf("Writer %d finishing work\n",myid);
return 0;
}

```

Приведем пример вывода *результатов работы программы* на терминал, см. рис. 33.

```

createConfig:
 NumOfreadrs = 3 ; ReadersDelay= 500 ; NumOfwriters= 4 ; WritersDelay = 200 ; si
zeofqueue = 5 ; ttl = 1
create readers
create writers
create TimeManager
successfully created threads
Writer 1 put data: "writer_id = 1 numMsg= 0" in position 0
Reader 1 get data: "writer_id = 1 numMsg= 0" from position 0
Writer 2 put data: "writer_id = 2 numMsg= 0" in position 1
Writer 3 put data: "writer_id = 3 numMsg= 0" in position 2
Reader 2 get data: "writer_id = 2 numMsg= 0" from position 1
Writer 0 put data: "writer_id = 0 numMsg= 0" in position 3
Writer 1 put data: "writer_id = 1 numMsg= 1" in position 4
Writer 2 put data: "writer_id = 2 numMsg= 1" in position 0
Writer 3 put data: "writer_id = 3 numMsg= 1" in position 1
Reader 2 get data: "writer_id = 3 numMsg= 0" from position 2
Reader 1 get data: "writer_id = 0 numMsg= 0" from position 3
Reader 0 get data: "writer_id = 1 numMsg= 1" from position 4
Writer 1 put data: "writer_id = 1 numMsg= 2" in position 2
Writer 2 put data: "writer_id = 2 numMsg= 2" in position 3
Writer 3 put data: "writer_id = 3 numMsg= 2" in position 4
Reader 2 get data: "writer_id = 2 numMsg= 1" from position 0
Writer 1 finishing work
TimeManager finishing work
Reader 0 finishing work
Reader 1 get data: "writer_id = 3 numMsg= 1" from position 1
Writer 3 finishing work
Writer 0 finishing work
Writer 2 finishing work
Reader 2 finishing work
Reader 1 finishing work
all is done

```

Рис. 33. Результат работы программы при синхронизации посредством мьютексов

Эксперимент проводился при следующих значениях параметров: число читателей = 3, число писателей = 4, задержка писателей – 200 мс, задержка читателей – 500 мс, размер очереди – 5, время работы программы – 1 с. Очевидно, что читатели работают быстрее, чем писатели, поэтому очередь довольно быстро заполняется. Присутствуют так же ситуации, когда очередь полна (строка 9 вывода читателей и писателей).

4.2. Семафоры для синхронизации в Windows

Сведения о средстве синхронизации

Объекты ядра «семафор» используются для учета ресурсов. Как и все объекты ядра, они содержат счетчик числа пользователей, но, кроме того, поддерживают два 32 битных значения со знаком: одно определяет максимальное число ресурсов (контролируемое семафором), другое используется как счетчик текущего числа ресурсов.

Для семафоров определены следующие правила:

- когда счетчик текущего числа ресурсов становится больше 0, семафор переходит в свободное состояние;
- если этот счетчик равен 0, семафор занят;
- система не допускает присвоения отрицательных значений счетчику текущего числа ресурсов;
- счетчик текущего числа ресурсов не может быть больше максимального числа ресурсов.

Объект ядра «семафор» создается вызовом **CreateSemaphore:**

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

PSECURITY_ATTRIBUTE psa – параметры защиты мьютекса;

LONG InitialCount – начальное значение мьютекса (число доступных изначально ресурсов);

LONG lMaximumCount – максимальное число ресурсов, обрабатываемое приложением (поскольку это 32-битное значение со знаком, предельное число ресурсов может достигать 2 147 483 647);

PCTSTR pszName – имя семафора.

Получить уже существующий семафор можно с помощью функции:

```
HANDLE OpenSemaphore( DWORD fdwAccess, BOOL bInheritHandle, PCTSTR  
pszName );
```

Параметры функции аналогичны функции **OpenMutex**.

Для получения доступа к ресурсу используются Wait-функции. Данные функции проверяют значение семафора (число свободных ресурсов) и, если оно больше 0, то потоку разрешается получить доступ (при этом число ресурсов семафора уменьшается на 1), иначе – нет.

Для освобождения ресурса используется функция:

```
BOOL ReleaseSemaphore( HANDLE hSem, LONG lReleaseCount,
PLONG lpPreviousCount );
```

HANDLE hSem – описатель семафора;

LONG lReleaseCount – значение, на которое необходимо увеличить число свободных ресурсов;

PLONG lpPreviousCount – возвращаемое предыдущее значение семафора.

Стоит отметить, что мьютекс – семафор с максимальным числом ресурсов равным 1.

Применение семафоров и изменения в исходном коде шаблона

Как и в случае мьютексов, в исходный код добавляется глобальная переменная-описатель семафора:

```
HANDLE sem;
```

Главный поток содержит функции удаления и создания семафора, потоки читатель и писатель – захвата и освобождения:

```
int main(int argc, char* argv[])
{
    char filename[30]; //имя файла конфигурации
    if(argc<2)
    {
        //используем конфигурацию по-умолчанию
        strcpy(filename,DEF_CONFIGFILE);
    }
    else
    {
        ...
        //инициализируем средство синхронизации
        sem>CreateSemaphore(NULL,1,1,""); // изначально семафор
        свободен

        ...
        for(int i=0;i<config.numOfReaders+config.numOfWriters+1;i++)
        //закрываем handle потоков
        CloseHandle(allhandlers[i]);
        //удаляем объект синхронизации
        CloseHandle(sem);
        printf("all is done\n");
        return 0;
    }
    DWORD WINAPI ThreadWriterHandler(LPVOID prm)
    {
        int myid=(int)prm;
        char tmp[50];
```

```

int msgnum=0;
while( isDone!=true)
{
    //Захват синхронизирующего объекта
    WaitForSingleObject(sem, INFINITE);
    ...
    queue.full=queue.writeindex==queue.readindex ? 1 : 0;
//если очередь заполнилась
}
// освобождение синхронизируемого объекта
ReleaseSemaphore(sem,1,NULL);
Sleep(config.writersDelay); //задержка
}
printf("Writer %d finishing work\n",myid);
return 0;
}
DWORD WINAPI ThreadReaderHandler(LPVOID prm)
{
    int myid=(int)prm;
    while( isDone!=true)
    {
        //Захват объекта синхронизации
        WaitForSingleObject(sem, INFINITE);
        ...
        queue.readindex=(queue.readindex+1)%queue.size;
    }
    //Освобождение объекта синхронизации
    ReleaseSemaphore(sem,1,NULL);
    Sleep(config.readersDelay); //задержка
}
printf("Reader %d finishing work\n",myid);
return 0;
}

```

Результаты выполнения

Для анализа результатов целесообразно произвести запуск с теми же параметрами, что и в предыдущем эксперименте.

В программе был использован семафор со значением 1, что аналогично использованию мьютекса. Поэтому, *потенциал данного средства синхронизации не был использован полностью*. Для полноценного использования было бы логичным в качестве ресурса рассматривать не очередь целиком, а ее отдельные элементы. Счетчик в этом случае может быть применен для подсчета записанных, но еще не считанных записей. Другой вариант использования семафоров – подсчет потоков, получающих доступ к ресурсу.

4.3. Критические секции

Сведения о средстве синхронизации

Критическая секция (critical section) — это небольшой участок кода, требующий монопольного доступа к каким-то общим данным. Она позволяет сделать так, чтобы одновременно только один поток получал доступ к определенному ресурсу.

Единственное, но важное преимущество критической секции по сравнению с мьютексом заключается в том, что обращение к ней происходит примерно в 100 раз быстрее, чем к объекту ядра. Связано это с тем, что

Критическая секция не является объектом ядра (она – объект пользовательского режима). Поэтому, с ней нельзя использовать WaitForMultipleObjects и другие функции ожидания. Вместо неё для критической секции есть своя специальная функция: EnterCriticalSection. По сравнению с WaitForMultipleObjects она предельно упрощена: ей можно передать указатель только на одну критическую секцию и нельзя даже указать таймаут. Функции ReleaseMutex тоже соответствует своя функция: LeaveCriticalSection. Таким образом, чтобы защитить участок кода, где ведётся работа с общими данными, от вмешательства других потоков, надо окружить его парой вызовов: EnterCriticalSection - LeaveCriticalSection.

Ниже в таблице представлено сравнение критической секции и мьютекса.

Таблица 4. Сравнение мьютексов и критических секций

Характеристики	Объект-мьютекс	Объект-критическая секция
Быстродействие	Малое	Высокое
Возможность использования за границами процесса	Да	Нет
Объявление	HANDLE mutex;	CRITICAL_SECTION crsect;
Инициализация	mutex=CreateMutex(..);	InitializeCriticalSection(&crsect);
Очистка	CloseHandle(mutex)	DeleteCriticalSection(&crsect);
Бесконечное ожидание	WaitForSingleObject(mutex, INFINITE);	EnterCriticalSection(&crsect);
Ожидание в течен. 0 мс	WaitForSingleObject(mutex,0);	TryEnterCriticalSection(&crsect);
Ожид. в течение произв. промежутка времени	WaitForSingleObject(mutex,time);	Невозможно
Освобождение	ReleaseMutex(mutex);	LeaveCriticalSection(&crsect);
Возможность параллельного ожидания других объектов ядра	Да (с помощью WaitForMultipleObjects или аналогичных)	Нет

Применение критических секций и изменения в исходном коде шаблона

Введем глобальную переменную:

```
CRITICAL_SECTION crs;
```

Главный поток инициализирует переменную, вызывая **InitializeCriticalSection**, после завершения работы он вызывает **DeleteCriticalSection**.

Потоки читатели и писатели захватывают секцию с помощью **EnterCriticalSection**, а затем освобождают с помощью **LeaveCriticalSection**.

Исходный код:

```
CRITICAL_SECTION crs;
```

```
int main(int argc, char* argv[])
{
    char filename[30]; //имя файла конфигурации
    if(argc<2)
    {
        //используем конфигурацию по-умолчанию
        strcpy(filename, DEF_CONFIGFILE);
    }
    else
    {
        ...
        //инициализируем средство синхронизации
        InitializeCriticalSection(&crs);

        //запускаем потоки на исполнение
        for(int i=0;i<config.numOfReaders+config.numOfWriters+1;i++)
            ResumeThread(allhandlers[i]);
        ...
        //удаляем объект синхронизации
        DeleteCriticalSection(&crs);

        printf("all is done\n");
        getchar();
        return 0;
    }

    DWORD WINAPI ThreadWriterHandler(LPVOID prm)
    {
        int myid=(int)prm;
        char tmp[50];
        int msgnum=0;
        while(isDone!=true)
        {
            //Захват синхронизирующего объекта
            EnterCriticalSection(&crs);

            if(queue.readindex!=queue.writeindex || !queue.full==1)
            //если в очереди есть место
            {
                ...
            }
        }
    }
}
```

```

        queue.full=queue.writeindex==queue.readindex ? 1 : 0;
//если очередь заполнилась
    }

    // освобождение синхронизируемого объекта

    LeaveCriticalSection(&crs);
    Sleep(config.writersDelay); //задержка

}
printf("Writer %d finishing work\n",myid);
return 0;
}
DWORD WINAPI ThreadReaderHandler(LPVOID prm)
{
    int myid=(int)prm;

    while(isDone!=true)
    {
        //Захват объекта синхронизации
        EnterCriticalSection(&crs);

        if(queue.readindex!=queue.writeindex || queue.full==1) //если
в очереди есть данные
        {
            ...
            queue.readindex=(queue.readindex+1)%queue.size;
        }

        //Освобождение объекта синхронизации
        LeaveCriticalSection(&crs);

        Sleep(config.readersDelay); //задержка

    }
    printf("Reader %d finishing work\n",myid);
    return 0;
}

```

4.4. Объекты-события в качестве средства синхронизации

Сведения о средстве синхронизации

События - самая примитивная разновидность объектов ядра. Они содержат счетчик числа пользователей (как и все объекты ядра) и две булевы переменные: одна сообщает тип данного объекта-события, другая — его состояние (свободен или занят).

События просто уведомляют об окончании какой-либо операции. Объекты-события бывают двух типов: со сбросом вручную (manual-reset events)

и с автосбросом (auto-reset events). Первые позволяют возобновлять выполнение сразу нескольких ждущих потоков, вторые — только одного.

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

Объект ядра «событие» создается функцией **CreateEvent**:

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

PSECURITY_ATTRIBUTES psa – атрибуты защиты;

BOOL fManualReset – режим переключения. TRUE – с автосбросом (после захвата потоком события, оно автоматически становится занятым), FALSE – без;

BOOL fInitialState – начальное состояние события. TRUE – свободное, FALSE – занятое;

PCTSTR pszName – имя события.

Функция возвращает описатель события.

Для ожидания события так же используются функции **Wait**.

Функции: `BOOL SetEvent(HANDLE hEvent)` и `BOOL ResetEvent(HANDLE hEvent)` позволяют установить событие в свободное и занятое состояние соответственно.

Удаление события производится функцией **CloseHandle**.

Применение и изменение в исходном коде шаблона

Главный поток создает и удаляет объект-событие. Потоки писатели и читатели ожидают освобождения события, захватывают и после освобождают его.

```
HANDLE event;
int main(int argc, char* argv[])
{

    char filename[30]; //имя файла конфигурации
    if(argc<2)
    {
        ...

        //инициализируем средство синхронизации
        event=CreateEvent(NULL, false, true, "");
    }
}
```



```

//запускаем потоки на исполнение
for(int i=0;i<config.numOfReaders+config.numOfWriters+1;i++)
    ResumeThread(allhandlers[i]);
...

for(int i=0;i<config.numOfReaders+config.numOfWriters+1;i++)
//закрываем handle потоков
    CloseHandle(allhandlers[i]);

//удаляем объект синхронизации
CloseHandle(event);

printf("all is done\n");
getchar();
return 0;
}

DWORD WINAPI ThreadReaderHandler(LPVOID prm)
{
    int myid=(int)prm;

    while(isDone!=true)
    {
        //Захват объекта синхронизации
        WaitForSingleObject(event,INFINITE);
        if(queue.readindex!=queue.writeindex || queue.full==1) //если в
очереди есть данные
        {
            queue.full=0;    //взяли данные, значит очередь не пуста
            ...
        }

        //Освобождение объекта синхронизации
        SetEvent(event);

        Sleep(config.readersDelay);    //задержка
    }
    printf("Reader %d finishing work\n",myid);

    return 0;
}

DWORD WINAPI ThreadWriterHandler(LPVOID prm)
{
    int myid=(int)prm;
    char tmp[50];
    int msgnum=0;
    while(isDone!=true)
    {

```

```

//Захват синхронизирующего объекта

WaitForSingleObject(event, INFINITE);

if(queue.readindex!=queue.writeindex || !queue.full==1) //если в
//очереди есть место
{

    //вносим в очередь данные
    ...
    queue.full=queue.writeindex==queue.readindex ? 1 : 0; //если
//очередь заполнилась
}

// освобождение синхронизируемого объекта
SetEvent(event);
Sleep(config.writersDelay); //задержка
}
printf("Writer %d finishing work\n",myid);
return 0;
}

```

4.5. Условные переменные

Описание средства синхронизации

Данное средство синхронизации само по себе не применяется, а только в сочетании с другими средствами синхронизации (например, с критическими секциями). Условные переменные отсутствуют в Windows 2003 и XP.

Для создания условной переменной используется функция:

```

VOID WINAPI InitializeConditionVariable(_Out_ PCONDITION_VARIABLE
ConditionVariable);

```

Функция ожидания освобождения условной переменной:

```

BOOL WINAPI SleepConditionVariableCS(
    _Inout_ PCONDITION_VARIABLE ConditionVariable, //условная переменная
    _Inout_ PCRITICAL_SECTION CriticalSection,
    //критическая секция, захваченная нами
    _In_ DWORD dwMilliseconds //время ожидания
);

```

Если функция вызвана успешно, то возвращается ненулевое значение.

Для пробуждения потоков, ожидающих условной переменной используются функции:

```

VOID WINAPI WakeAllConditionVariable(_Inout_ PCONDITION_VARIABLE
ConditionVariable);

```

и

```

VOID WINAPI WakeConditionVariable(_Inout_ PCONDITION_VARIABLE
ConditionVariable);

```

которые пробуждают все ожидающие потоки, либо только один.

Изменение в исходном коде шаблона

В код были введены 3 глобальные переменные: критическая секция, условная переменная на запись, условная переменная на чтение.

Критическая секция позволяет только одному потоку получить доступ к ресурсу-очереди. Условная переменная на запись используется для пробуждения одного из потоков-писателей, условная переменная на чтение – одного из потоков-читателей.

В отличие от предыдущих примеров, в данном случае потоки периодически не опрашивают очередь на предмет её освобождения, т.к. они просыпаются с уже освобожденной либо занятой чем-либо очередью.

Исходный код:

```
CRITICAL_SECTION crs;//критическая секция общая и для писателей и для читателей
CONDITION_VARIABLE condread; //условная переменная для потоков-писателей
CONDITION_VARIABLE condwrite; //условная переменная для потоков-читателей

int main(int argc, char* argv[])
{
    char filename[30]; //имя файла конфигурации
    if(argc<2)
    {
        ...
        queue.data=new char*[config.sizeOfQueue];

        //инициализируем средство синхронизации
        InitializeCriticalSection(&crs);
        InitializeConditionVariable(&condread);
        InitializeConditionVariable(&condwrite);

        //запускаем потоки на исполнение
        for(int i=0;i<config.numOfReaders+config.numOfWriters+1;i++)
            ResumeThread(allhandlers[i]);

        //ожидаем завершения всех потоков
        WaitForMultipleObjects(config.numOfReaders+config.numOfWriters+1,all
handlers,TRUE,5000);

        for(int i=0;i<config.numOfReaders+config.numOfWriters+1;i++)
//закрываем handle потоков
            CloseHandle(allhandlers[i]);

        //удаляем объект синхронизации
        DeleteCriticalSection(&crs);

        printf("all is done\n");
        getchar();
    }
}
```

```

    return 0;
}

DWORD WINAPI ThreadWriterHandler(LPVOID prm)
{
    int myid=(int)prm;
    char tmp[50];
    int msgnum=0;
    while(isDone!=true)
    {
        //Захват синхронизирующего объекта

        EnterCriticalSection(&crs);

        while(!(queue.readindex!=queue.writeindex ||
!queue.full==1))
            //спим пока в очереди не освободится место
            SleepConditionVariableCS(&condwrite,&crs,INFINITE);

        //заносим в очередь данные
        sprintf(tmp,"writer_id = %d numMsg= %3d",myid,msgnum);
        ...
        if(queue.full==1)
            printf("queue is full\n");
        //шлем сигнал потокам-читателям
        WakeConditionVariable(&condread);
        // освобождение синхронизируемого объекта
        LeaveCriticalSection(&crs);

        Sleep(config.writersDelay); //задержка
    }
    printf("Writer %d finishing work\n",myid);
    return 0;
}

DWORD WINAPI ThreadReaderHandler(LPVOID prm)
{
    int myid=(int)prm;
    while(isDone!=true)
    {
        //Захват объекта синхронизации
        EnterCriticalSection(&crs);

        while(!(queue.readindex!=queue.writeindex ||
queue.full==1))
            //спим, пока в очереди не появятся данные
            SleepConditionVariableCS(&condread,&crs,INFINITE);

        queue.full=0; //взяли данные, значит очередь не пуста
        ...
        queue.readindex=(queue.readindex+1)%queue.size;

        //отправляем сигнал потокам-читателям
    }
}

```

```

WakeConditionVariable(&condwrite);
// освобождение синхронизируемого объекта
LeaveCriticalSection(&crs);

Sleep(config.readersDelay); //задержка
}
printf("Reader %d finishing work\n",myid);
return 0;
}

```

Так же стоит отметить, что завершить все потоки стало сложнее. Т.к. они могут навсегда зависнуть в режиме ожидания. Применение установки таймаута на завершение потоков может решить эту проблему.

4.6. Функции ожидания

В работе использовались функции **WaitForSingleObject** и **WaitForMultipleObject**, которые основывались на проверке состояния объекта (занят он или свободен). В некоторых случаях данные функции не только проверяют состояние, но и при свободном объекте сбрасывает его в занятое состояние.

4.7. Задача «читатели и писатели»

Пример 2. Задача «читатели и писатели»

Рассмотрим частный случай этой задачи для демонстрации использования **объектов-событий для синхронизации доступа к памяти.**

Задание.

Необходимо решить задачу одного писателя и N читателей. Для синхронизации разрешено использовать только объекты-события, в качестве разделяемого ресурса – разделяемую память (share memory). Писатель пишет в share memory сообщение и ждет, пока все читатели не прочитают данное сообщение.

Задача должна быть решена сначала для потоков, принадлежащих одному процессу, а затем – разным независимым процессам.

2.1. Решение задачи для потоков одного процесса

Описание программы

Пусть количество потоков-читателей известно и не изменяется со временем. Тогда общее число потоков всегда будет равно: 1+1+N (главный поток, поток-писатель, потоки-читатели соответственно).

Общий алгоритм работы с разделяемой памятью (по шагам):

- 1) Создать объект отображения (или получить дескриптор уже созданного);
- 2) Отобразить файла на АП процесса (получить указатель на разделяемую память);
- 3) Работать с памятью;
- 4) После завершения – убрать отображение файла на АП процесса и закрыть дескриптор объекта-отображения.

Для создания объекта-отображения используется функция:

```
HANDLE CreateFileMapping(HANDLE hFile,  
LPSECURITY_ATTRIBUTES lpSa, DWORD dwProtect, DWORD  
dwMaximumSizeHigh, DWORD dwMaximumSizeLow, LPCTSTR  
lpMapName )
```

HANDLE hFile — дескриптор открытого файла, атрибуты защиты которого совместимы с флагами защиты, указанными параметром **dwProtect**. Значение этого дескриптора (тип данных HANDLE), равное 0xFFFFFFFF (его эквивалент — символическая константа INVALID_HANDLE_VALUE), соответствует системному файлу подкачки, и его можно использовать для организации разделения памяти несколькими процессами без создания отдельного файла;

LPSECURITY_ATTRIBUTES lpSa – атрибуты защиты объекта-отображения;

DWORD dwProtect – определяет возможности доступа к представлению файла при отображении (используются флаги PAGE_READONLY, PAGE_READWRITE и другие);

DWORD dwMaximumSizeHigh, DWORD dwMaximumSizeLow – старшая и младшая 32-битовые части значения максимального размера объекта отображения файла. Данное значение впоследствии измениться не может. Если оба параметра равны 0, то используется размер файла, дескриптор которого передан через параметр **hFile**;

LPCTSTR lpMapName – имя объекта-отображения.

Возвращаемое значение: в случае успешного выполнения — дескриптор объекта отображения файла, иначе — NULL.

Для получения дескриптора уже созданного объекта-отображения используется функция:

```
HANDLE OpenFileMapping(DWORD dwDesiredAccess, BOOL  
bInheritHandle, LPCTSTR lpMapName)
```

DWORD dwDesiredAccess – тот же набор флагов, что и у параметра **DWORD dwProtect** в функции `CreateFileMapping`.

BOOL bInheritHandle – дескриптор может быть унаследован;

LPCTSTR lpMapName – имя объекта-отображения (соответствует параметру **LPCTSTR lpMapName** в функции `CreateFileMapping`).

Возвращаемое значение: в случае успешного выполнения — дескриптор объекта отображения файла, иначе — `NULL`.

Отображение файла на АП процесса осуществляется с помощью функции:

```
LPVOID MapViewOfFile(HANDLE hMapObject, DWORD dwAccess,  
DWORD dwOffsetHigh, DWORD dwOffsetLow, SIZE_T cbMap)
```

HANDLE hMapObject – дескриптор объекта-отображения файла (возвращенного функцией `CreateFileMapping` или `OpenFileMapping`);

DWORD dwAccess – параметр доступа к файлу (должен совпадать с правами доступа к объекту-отображению). Принимает значения: `FILE_MAP_WRITE`, `FILE_MAP_READ` и `FILE_MAP_ALL_ACCESS`;

DWORD dwOffsetHigh, DWORD dwOffsetLow - соответственно, старшая и младшая 32-битовые части смещения начала отображаемого участка в файле. Значение этого начального адреса должно быть кратным 64 Кбайт. Чтобы начало отображаемого участка совпадало с началом файла, оба параметра следует задать равными 0.

SIZE_T cbMap - размер отображаемого участка файла в байтах. Если значение этого параметра установлено равным 0, то отображаться будет весь файл, существующий в момент вызова функции `MapViewOfFile`.

Возвращаемое значение: в случае успешного выполнения — начальный адрес блока (представления файла), иначе — `NULL`.

Для очистки памяти («удаления» отображения) используется функция:

```
BOOL UnmapViewOfFile(LPVOID lpBaseAddress)
```

LPVOID lpBaseAddress – указатель на область памяти, который был возвращен функцией `MapViewOfFile`.

Возвращаемое значение: в случае успешного выполнения — ненулевое значение, иначе — NULL.

В программе используются следующие глобальные переменные:

```
//глобальные переменные
struct Configuration config; //конфигурация программы
bool isDone=false; //флаг завершения
HANDLE *allhandlers; //массив всех создаваемых потоков

HANDLE
canReadEvent, canWriteEvent, changeCountEvent, exitEvent, allReadEvent
; //события для синхронизации
int countread=0, countready=0; //переменные для синхронизации
работы потоков
char shareFileName[]="$MyVerySpecialShareFileName$"; //имя
разделяемой памяти
HANDLE hFileMapping; //объект-отображение файла
LPVOID lpFileMapForWriters, lpFileMapForReaders; // указатели на
отображаемую память
```

Для синхронизации потоков необходимо 5 объектов-событий:

- *CanReadEvent* – событие «можно читать». Означает, что поток-писатель записал сообщение в память, и его могут читать потоки-читатели (ручной сброс);
- *CanWriteEvent* – событие «можно писать». Означает, что все потоки-читатели получили сообщение и готовы к приему следующего (автосброс);
- *allReadEvent* – событие «все потоки-читатели прочитали сообщение». Требуется для приведения в готовность потоков-читателей (см. описание алгоритма работы потоков-читателей ниже) (ручной сброс);
- *changeCountEvent* – событие для разрешения работы со счетчиком (количество потоков-читателей, которые прошли заданный участок кода) (автосброс);
- *exitEvent* – событие «завершение программы». Устанавливается потоком-планировщиком (ручной сброс).

Основная логика работы программы, по сравнению с пунктами 3.1. и т.д. не изменилась (не изменялось содержимое файла `utils.cpp`).

Главный поток – осуществляет инициализацию всех ресурсов, создание и запуск потоков, а после завершения последних – очистку. Схема алгоритма работы представлена на рис. 34.

Разделяемая память создается через файл подкачки.



Рис. 34. Схема алгоритма работы главного потока в примере с использованием объектов-событий для синхронизации доступа к памяти потоков одного процесса

Исходный код функции главного потока:

```

int main(int argc, char* argv[])
{
    char filename[30]; //имя файла конфигурации
    if(argc<2)
    {
        //используем конфигурацию по-умолчанию
        strcpy(filename,DEF_CONFIGFILE);
    }
    else
    {
        strcpy(filename,argv[1]);
    }

    ////функция установки конфигурации (передаем имя читаемого
//файла и заполняем структуру)
    SetConfig(filename,&config);
    ////создаем необходимые потоки без их запуска
    CreateAllThreads(&config); //потоки-читатели запускаются сразу
// (чтобы они успели дойти до функции ожидания)
  
```

```

//Инициализируем ресурс (share memory)
if((hFileMapping=CreateFileMapping(INVALID_HANDLE_VALUE,NULL,
PAGE_READWRITE,0,1500,shareFileName))==NULL) //создаем объект
//"отображаемый файл"
//т.к. в качестве дескриптора файла использовано значение
равное 0xFFFFFFFF (его эквивалент – символическая константа
INVALID_HANDLE_VALUE)
//то будет использован системный файл подкачки, при этом на
диске файл создаваться не будет
{
printf("impossible to create shareFile\n Last error %d\n",GetLastError());
ExitProcess(10000);
}

//отображаем файл на адресное пространство нашего процесса для потока-писателя
lpFileMapForWriters=MapViewOfFile(hFileMapping,FILE_MAP_WRITE,
0,0,0); // (с начала файла, размером = размеру файла)

//отображаем файл на адресное пространство нашего процесса для потоков-читателей
lpFileMapForReaders=MapViewOfFile(hFileMapping,FILE_MAP_READ,0,0,0);
// (с начала файла, размером = размеру файла)

//инициализируем средства синхронизации
canReadEvent=CreateEvent(NULL,true,false,""); //событие "окончание
// записи" ("можно читать"), ручной сброс, изначально занято
canWriteEvent=CreateEvent(NULL,false,false,""); //событие -
//"можно писать", автосброс (разрешаем писать только одному),
//изначально свободно
allReadEvent=CreateEvent(NULL,true,true,""); //событие "все
//прочитали"
changeCountEvent=CreateEvent(NULL,false,true,""); //событие для
//изменения счетчика (сколько клиентов еще не прочитало сообщение)
exitEvent=CreateEvent(NULL,true,false,""); //событие
//"завершение работы программы", ручной сброс, изначально занято

//запускаем потоки-писатели и поток-планировщик на исполнение
for(int i=0;i<config.numOfReaders+config.numOfWriters+1;i++)
ResumeThread(allhandlers[i]);

//ожидаем завершения всех потоков

WaitForMultipleObjects(config.numOfReaders+config.numOfWriters+1,a
llhandlers,TRUE,INFINITE);

for(int i=0;i<config.numOfReaders+config.numOfWriters+1;i++)
//закрываем handle потоков
CloseHandle(allhandlers[i]);

//закрываем описатели объектов синхронизации
CloseHandle(canReadEvent);
CloseHandle(canWriteEvent);

```

```

CloseHandle(allReadEvent);
CloseHandle(changeCountEvent);
CloseHandle(exitEvent);

//закрываем handle общего ресурса
UnmapViewOfFile(lpFileMapForReaders);
UnmapViewOfFile(lpFileMapForWriters);
CloseHandle(hFileMapping); //закрываем объект "отображаемый файл"
printf("all is done\n");
getchar();
return 0;
}

```

Поток-координатор выполняет такую же работу, как и раньше (см. описание работы в пункте 3.1.2). Единственное, теперь вместе с установкой флага `IsDone` (при завершении работы программы) так же освобождается событие `exitEvent`.

Схема алгоритма работы потока-писателя представлена на рис. 35.

Поток-писатель ждет, пока все потоки-читатели прочитают сообщение, а затем пишет новое сообщение. После записи он разрешает чтение.

Поток-писатель работает с 2 объектами-событиями: `canWriteEvent` и `canReadEvent`, так же он устанавливает количество потоков-читателей, которые должны прочитать записанную им информацию (в программе с потоками разных процессов это значение будет храниться в разделяемой памяти).

Счетчик декрементируется каждый раз, когда поток-читатель прочитал данные, это позволяет следить за тем, чтобы все потоки-читатели прочитали записанную информацию.

Исходный код потока-писателя:

```

DWORD WINAPI ThreadWriterHandler(LPVOID prm)
{
    int myid=(int)prm;
    //char tmp[50];
    int msgnum=0;
    HANDLE writerhandlers[2];
    writerhandlers[0]=exitEvent;
    writerhandlers[1]=canWriteEvent;

    DWORD dwEvent; //возвращаемое значение для waitformultipleobjects

    while(isDone!=true)
    {

        dwEvent=WaitForMultipleObjects(2,writerhandlers,false,INFINITE
    );
        switch(dwEvent)

```

```

{
    case WAIT_OBJECT_0: //сработало событие exit
        printf("Writer %d finishing work\n",myid);
        return 0;
    case WAIT_OBJECT_0+1: // сработало событие на возможность записи
        msgnum++; //увеличиваем номер сообщения
        countread=config.numOfReaders;
        sprintf((char *) lpFileMapForWriters,"writer_id %d,
msg with num = %d",myid,msgnum);
        printf("writer put msg: \"%s\" \n",lpFileMapForWriters);
        SetEvent(canReadEvent); //разрешаем потокам - читателям
//прочитать сообщение и возвращаем событие в занятое
        break;
    default:
        printf("error with func WaitForMultipleObjects in
writerHandle\n");
        printf("getlasterror= %d\n",GetLastError());
        ExitProcess(1000);
}
}
printf("Writer %d finishing work\n",myid);
return 0;
}

```

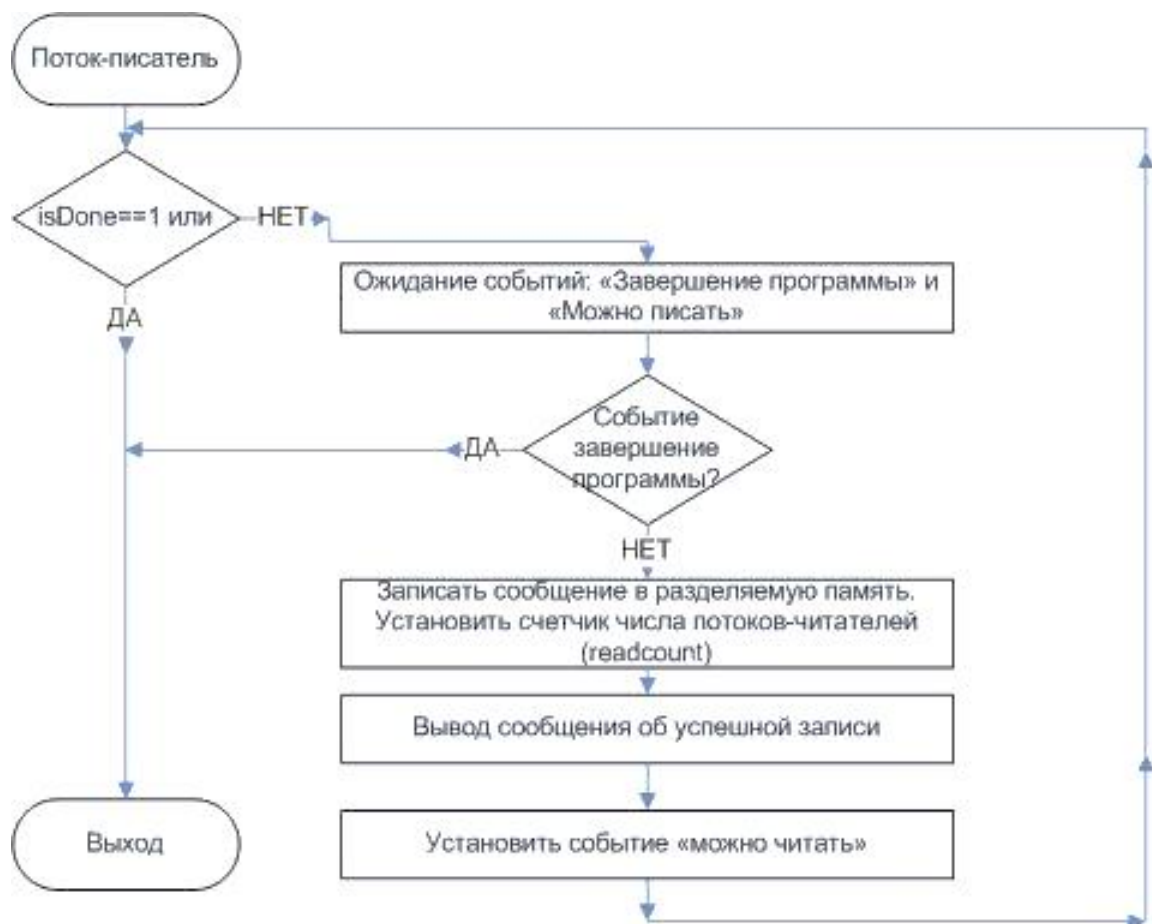


Рис. 35. Схема алгоритма работы потока-писателя (пример 2)

Потоки-читатели работают по более сложному алгоритму. Для их синхронизации необходимо использовать 4 объекта-события (вместо двух у писателя): *canReadEvent*, *canWriteEvent*, *allReadEvent*, *changeCountEvent*. Пояснить работу проще всего по схеме алгоритма, представленной на рис. 36.

Для того, чтобы определить, прошли все потоки-читатели участок кода или нет, используются переменные *readcount* и *readready*. *Readcount* – определяет число потоков, которое уже прочитали данные, *readready* – число потоков, готовых для чтения сообщения и ожидающих только сигнала от потока-писателя.

Последний поток-читатель, проходящий «критические участки», производит «закрывание дверей за собой». Это связано с тем, что для синхронизации используются события с ручным сбросом и мы не можем знать, когда все потоки-читатели пройдут «критический участок» и будут готовы к приему новых данных.

Рассмотрим следующую ситуацию. Пусть все потоки-читатели ожидают события «можно читать». При этом «закрыто» событие «все прочитали», открыто событие «работа со счетчиком». Пусть событие «можно читать» происходит (его устанавливает поток-писатель). При этом оно автоматически не сбрасывается в занятое состояние (ручной сброс). Все потоки-читатели одновременно читают общую память, выводят сообщение, а затем по одному изменяют счетчик *readcount*, установленный потоком-писателем. Далее все потоки «скапливаются» у ожидания события «все прочитали» (т.к. пока оно заблокировано).

Последний поток «закрывает» событие «можно читать» и «открывает» событие «все прочитали». Тогда все потоки-читатели начинают «скапливаться» у ожидания события «можно читать». При этом потоки изменяют счетчик *readready* (число готовых потоков). Последний из потоков-читателей обнуляет данный счетчик, «закрывает» событие «все прочитали» и устанавливает событие «можно писать». Далее работу начинает поток-писатель и цикл повторяется.

Если бы для синхронизации потоков-читателей использовалось только событие «можно читать», то могла встретиться ситуация, что 1 поток мог прочитать 2 раза (т.к. он успел пойти на второй цикл чтения).

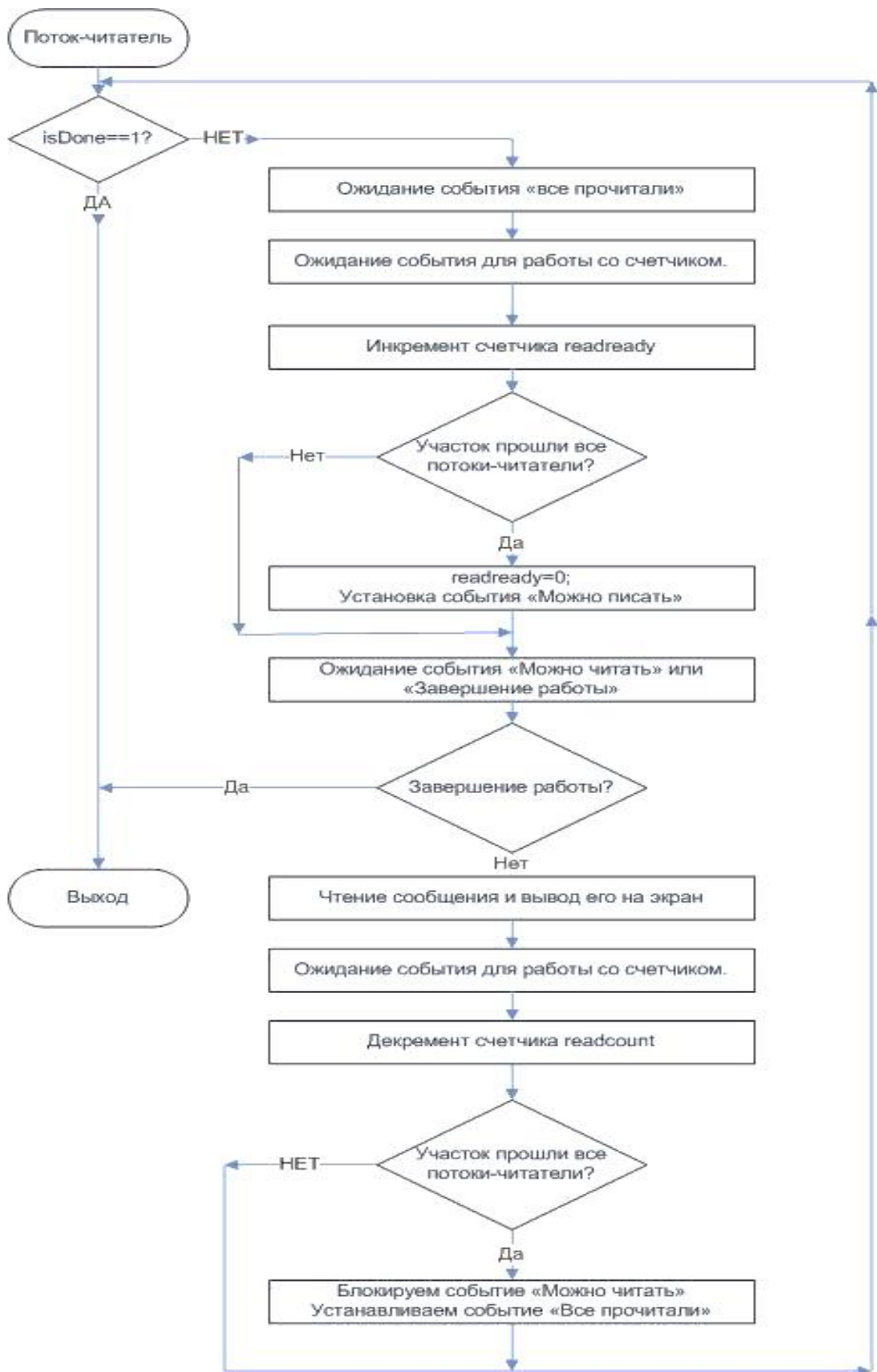


Рис. 36. Схема алгоритма работы потока-читателя в примере с использованием объектов-событий для синхронизации доступа к памяти для потоков одного процесса

Исходный код потока-читателя:

```
DWORD WINAPI ThreadReaderHandler(LPVOID prm)
{
    int myid=(int)prm;
    HANDLE readerhandlers[2];
    readerhandlers[0]=exitEvent;
    readerhandlers[1]=canReadEvent;
    DWORD dwEvent; //возвращаемое значение для
waitformultipleobjects

    while(isDone!=true)
    {
        WaitForSingleObject(allReadEvent,INFINITE); //ждем, пока все
//прочитают
        //узнаем, сколько потоков-читателей прошло данную границу
        WaitForSingleObject(changeCountEvent,INFINITE);
        countready++;
        if (countready==config.numOfReaders)
        {
            countready=0;
            ResetEvent(allReadEvent); //если все прошли, то
//"закрываем за собой дверь"
            SetEvent(canWriteEvent); //и разрешаем писать
        }
        SetEvent(changeCountEvent); //разрешаем изменять счетчик

        dwEvent=WaitForMultipleObjects(2,readerhandlers,false,INFINITE
);
        switch(dwEvent)
        {
            case WAIT_OBJECT_0: //сработало событие exit
                printf("Reader %d finishing work\n",myid);
                return 0;
            case WAIT_OBJECT_0+1: // сработало событие на возможность чтения

                printf("Reader %d read msg \"%s\"\n",myid,(char *)
lpFileMapForReaders); //читаем сообщение

                //необходимо уменьшить счетчик количества читателей,
//которые прочитать еще не успели
                WaitForSingleObject(changeCountEvent,INFINITE);
                countread--;
                if(countread==0)
                {
                    ResetEvent(canReadEvent); // если мы последние
//читали, то запрещаем читать
                    SetEvent(allReadEvent); //открываем границу
                }
                SetEvent(changeCountEvent);
                break;
            default:
```

```

        printf("error with func WaitForMultipleObjects in
readerHandle\n");
        printf("getlasterror= %d\n",GetLastError());
        ExitProcess(1001);

    }
}
printf("Reader %d finishing work\n",myid);
return 0;
}

```

В результате выполнения программы новое сообщение не записывается, пока предыдущее не прочтется всеми потоками-читателями. Порядок чтения для потоков-читателей произволен.

Решение задачи «читатели-писатели» для потоков разных процессов с синхронизацией объектами

Описание программы

В данной программе главный поток и поток-писатель будут принадлежать одному процессу, **потоки-читатели – разным**. Главный процесс создает процессы-читатели и 2 потока: писатель и планировщик. Для наглядности каждый процесс-читатель связан со своей консолью.

Каждому объекту-событию теперь присваивается имя: “\$My_имя_события\$”.

Так же для потоков-читателей теперь невозможно остановиться по флагу `isDone` (они завершаются только по событию `exitEvent`).

Счетчики `readcount` и `readready` не доступны напрямую потокам-читателям. Поэтому данные счетчики пишутся в разделяемую память. В начале 4 байта занимает `readcount`, затем 4 байта – `readready`, остальное – передаваемое писателем сообщение. При этом для читающих потоков доступ к разделяемой памяти устанавливается как на чтение, так и на запись.

Единственное изменение в алгоритме произошло у потока-писателя. Теперь данный поток устанавливает оба счетчика (ранее только `readcount`).

Исходный код главного процесса:

```

#define _CRT_SECURE_NO_WARNINGS //чтобы не получать предупреждений
//по функциям strcpy, fopen etc.

#include <stdio.h>
#include <Windows.h>
#include "utils.h"

```



```

#define DEF_CONFIGFILE "default"

//глобальные переменные
struct Configuration config; //конфигурация программы
bool isDone=false;
HANDLE *allhandlers; //массив всех создаваемых потоков

    //события для синхронизации
HANDLE
canReadEvent,canWriteEvent,changeCountEvent,exitEvent,allReadEvent;
int countread=0,countready=0; //переменные для синхронизации работы потоков
char shareFileName[]="$MyVerySpecialShareFileName$"; //имя
//разделяемой памяти
HANDLE hFileMapping;
LPVOID lpFileMapForWriters; // указатель на отображаемую память

int main(int argc, char* argv[])
{
    char filename[30]; //имя файла конфигурации
    if(argc<2)
    {
        //используем конфигурацию по-умолчанию
        strcpy(filename,DEF_CONFIGFILE);
    }
    else
    {
        strcpy(filename,argv[1]);
    }

    ///функция установки конфигурации (передаем имя читаемого
//файла и заполняемую структуру)
    SetConfig(filename,&config);
    ///создаем необходимые потоки без их запуска
    CreateAllThreads(&config); //потоки-читатели запускаются сразу
// (чтобы они успели дойти до //функции ожидания)
    //Инициализируем ресурс (share memory)
    //создаем объект "отображаемый файл"
    if((hFileMapping=CreateFileMapping(INVALID_HANDLE_VALUE,NULL,
PAGE_READWRITE,0,1500,shareFileName)) ==NULL)
//т.к. в качестве дескриптора файла использовано значение равное 0xFFFFFFFF //(его
эквивалент — символическая константа INVALID_HANDLE_VALUE)
//то будет использован системный файл подкачки, при этом на диске
//файл создаваться не будет
    {
        printf("impossible to create shareFile\n Last error
%d\n",GetLastError());
        ExitProcess(10000);
    }

    //отображаем файл на адресное пространство нашего процесса для
потока-писателя
    //(от начала файла, размером, равным размеру файла)

```

```

    lpFileMapForWriters=MapViewOfFile(hFileMapping,FILE_MAP_WRITE,
0,0,0);        //инициализируем 2 переменные в общей памяти
(readready и readcount)
    *((int *)lpFileMapForWriters)=0;
    *((int *)lpFileMapForWriters)+1)=config.numOfReaders;

    //инициализируем средства синхронизации

//событие "окончание записи" ("можно читать"), ручной сброс, изначально занято
canReadEvent=CreateEvent(NULL,true,false,"$$My_canReadEvent$$");

//событие - "можно писать", автосброс (разрешаем писать только
//одному), изначально-свободно
    canWriteEvent=CreateEvent(NULL,false,false,"$$My_canWriteEvent$$");

//событие "все прочитали"
    allReadEvent=CreateEvent(NULL,true,true,"$$My_allReadEvent$$")
;
//событие для изменения счетчика (сколько клиентов еще не прочитало сообщение)
    changeCountEvent=CreateEvent(NULL,false,true,"$$My_changeCount
Event$$");
//событие "завершение работы программы", ручной сброс, изначально занято
    exitEvent=CreateEvent(NULL,true,false,"$$My_exitEvent$$");

//запускаем потоки-писатели и поток-планировщик на исполнение
    for(int i=0;i<config.numOfReaders+config.numOfWriters+1;i++)
        ResumeThread(allhandlers[i]);

    //ожидаем завершения всех потоков

WaitForMultipleObjects(config.numOfReaders+config.numOfWriters+1,a
llhandlers,TRUE,INFINITE);

for(int i=0;i<config.numOfWriters+1;i++) //закрываем описатели потоков
    CloseHandle(allhandlers[i]);

    //закрываем HANDLE объектов синхронизации
    CloseHandle(canReadEvent);
    CloseHandle(canWriteEvent);
    CloseHandle(allReadEvent);
    CloseHandle(changeCountEvent);
    CloseHandle(exitEvent);

    //закрываем handle общего ресурса
    UnmapViewOfFile(lpFileMapForWriters);
    CloseHandle(hFileMapping);        //закрываем объект "отображаемый
файл"
    printf("all is done\n");
    getchar();
    return 0;
}

```

```

DWORD WINAPI ThreadWriterHandler(LPVOID prm) //функция потока-
//планировщика
{
    int myid=(int)prm;
    //char tmp[50];
    int msgnum=0;
    HANDLE writerhandlers[2];
    writerhandlers[0]=exitEvent;
    writerhandlers[1]=canWriteEvent;
    DWORD dwEvent; //возвращаемое значение для
waitformultipleobjects
    while(isDone!=true)
    {

        dwEvent=WaitForMultipleObjects(2,writerhandlers,false,INFINITE
);
        switch(dwEvent)
        {
            case WAIT_OBJECT_0: //сработало событие exit
                printf("Writer %d finishing work\n",myid);
                return 0;
            case WAIT_OBJECT_0+1: // сработало событие на возможность записи
                msgnum++; //увеличиваем номер сообщения

                sprintf(((char *)
lpFileMapForWriters)+sizeof(int)*2,"writer_id %d, msg with num =
%d",myid,msgnum);
                printf("writer put msg: \"%s\" \n",((char *)
lpFileMapForWriters)+sizeof(int)*2);

                WaitForSingleObject(changeCountEvent,INFINITE);
                *((int *)lpFileMapForWriters)+=config.numOfReaders;
                *(((int *)lpFileMapForWriters)+1)+=config.numOfReaders;
                SetEvent(changeCountEvent);
//разрешаем потокам-читателям прочитать сообщение и опять ставим
//событие в состояние «занято»
                SetEvent(canReadEvent);
                break;

            default:
                printf("error with func WaitForMultipleObjects in
writerHandle\n");
                printf("getlasterror= %d\n",GetLastError());
                ExitProcess(1000);
        }
    }
    printf("Writer %d finishing work\n",myid);
    return 0;
}

DWORD WINAPI ThreadTimeManagerHandler(LPVOID prm) //поток-координатор
{
    int ttl=(int) prm;

```

```

if(ttl<0)
{
    char buf[100]; //завершение по команде оператора
    while(1)
    {
        fgets(buf,sizeof(buf),stdin);
        if(buf[0]=='s')
        {
            SetEvent(exitEvent);
            isDone=true;
            break;
        }
    }
}
else
{
    HANDLE h = CreateAndStartWaitableTimer(ttl); //завершение по
//таймеру
    WaitForSingleObject(h,INFINITE);
    SetEvent(exitEvent);
    isDone = true;
    CloseHandle(h);
}
printf("TimeManager finishing work\n");
return 0;
}

```

В файл `utils.cpp` вносим изменения. Вместо функции *CreateThread()* для потоков-читателей теперь вызывается *CreateProcess()*. При этом поток `main` сохраняет только идентификаторы потоков новых процессов-читателей.

Изменения в исходном коде `utils.cpp` (изменилась только функция *CreateAllThreads()*):

```

void CreateAllThreads(struct Configuration * config) //создание
//всех потоков
{
    printf("createConfig:\n NumOfreadrs = %d | ReadersDelay= %d |
NumOfwriters= %d | WritersDelay = %d | sizeofqueue = %d | ttl =
%d\n",config->numOfReaders,config->readersDelay,config-
>numOfWriters,config->writersDelay,config->sizeOfQueue,config-
>ttl);
    allhandlers=new HANDLE[config->numOfReaders+config-
>numOfWriters+1];
    int count=0;

    printf("create readers process\n");

    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory( &si, sizeof(si) );

```

```

    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );

    TCHAR szCommandLine[100];

    for(int i=0;i<=config->numOfReaders-1;i++, count++) //создаем
процессы-читатели
    {

        sprintf(szCommandLine,"..\\..\\ReadProcess\\Release\\ReadProce
ss.exe %d",i);
        printf("count= %d\n",count);
        CreateProcess(NULL, szCommandLine, NULL, NULL, FALSE,
CREATE_NEW_CONSOLE | CREATE_SUSPENDED, NULL, NULL, &si, &pi);
        allhandlers[count]=pi.hThread;
    }

    printf("create writers\n");
    for(int i=0;i<=config->numOfWriters-1;i++, count++) //создаем
потоки-писатели
    {
        ...
        printf("successfully created threads\n");
        return;
    }
}

```

Исходный код процесса-читателя:

```

#define _CRT_SECURE_NO_WARNINGS //чтобы не получать предупреждений
//по функциям strcpy, fopen etc.

#include <stdio.h>
#include <Windows.h>

int main(int argc, char* argv[])
{
    if(argc!=2) //проверяем число аргументов
    {
        printf("error with start reader process. Need 2 arguments\n");
        getchar();
        ExitProcess(1000);
    }
    //получаем из командной строки наш номер
    int myid=atoi(argv[1]);

    printf("reader with id= %d is started\n",myid);

    //получаем объекты синхронизации и общий ресурс
    HANDLE
canReadEvent,canWriteEvent,changeCountEvent,exitEvent,allReadEvent;
//события для синхронизации
canReadEvent=OpenEvent(EVENT_ALL_ACCESS,false,"$$My_canReadEvent$$");
//событие "окончание записи" ("можно читать"), ручной сброс, изначально занято
canWriteEvent=OpenEvent(EVENT_ALL_ACCESS,false,

```

```

        "$$My_canWriteEvent$$"); //событие - "можно
//писать", автосброс (разрешаем писать только одному), изначально свободно
allReadEvent=OpenEvent(EVENT_ALL_ACCESS,false,"$$My_allReadEvent$$");
//событие "все прочитали"
    changeCountEvent=OpenEvent(EVENT_ALL_ACCESS,false,
"$$My_changeCountEvent$$"); //событие для изменения счетчика
//(сколько клиентов еще не прочитало сообщение)
exitEvent=OpenEvent(EVENT_ALL_ACCESS,false,"$$My_exitEvent$$");

//общий ресурс
HANDLE hFileMapping;
hFileMapping=OpenFileMapping(FILE_MAP_ALL_ACCESS,false,"$$MyVerySpecialShareFileName$$");

//если объекты не созданы, то не сможем работать
if(canReadEvent==NULL || canWriteEvent==NULL || allReadEvent==
NULL || changeCountEvent==NULL || exitEvent==NULL ||
hFileMapping==NULL)
{
    printf("impossible to open objects, run server first\n
getlasterror= %d",GetLastError());
    getchar();
    return 1001;
}

//отображаем файл на адресное пространство нашего процесса для
//потокoв-читателей
LPVOID lpFileMapForReaders=MapViewOfFile(hFileMapping,
FILE_MAP_ALL_ACCESS,0,0,0); // (с начала файла,
размером равным размеру файла)

HANDLE readerhandlers[2];
readerhandlers[0]=exitEvent;
readerhandlers[1]=canReadEvent;
DWORD dwEvent; //возвращаемое значение для
waitformultipleobjects
//основной цикл
while(1)
{
    WaitForSingleObject(allReadEvent,INFINITE); //ждем, пока все
//прочитают
    //узнаем, сколько потокoв-читателей прошло данную границу
    WaitForSingleObject(changeCountEvent,INFINITE);
    //
    (((int *) lpFileMapForReaders)+1)--;
    printf("readready= %d\n",(((int *)
lpFileMapForReaders)+1));
    if (((int *) lpFileMapForReaders)+1)==0)
    {
        ResetEvent(allReadEvent); //если все прошли, то
//"закрываем за собой дверь"
        SetEvent(canWriteEvent); //и разрешаем писать
    }
}

```

```

SetEvent(changeCountEvent);

dwEvent=WaitForMultipleObjects(2,readerhandlers,false,INFINITE
);
switch(dwEvent)
{
case WAIT_OBJECT_0: //сработало событие exit
    printf("Reader %d finishing work\n",myid);
    goto exit;
case WAIT_OBJECT_0+1: // сработало событие на возможность чтения

    printf("Reader %d read msg \"%s\"\n",myid,((char *)
lpFileMapForReaders)+sizeof(int)*2); //читаем сообщение

    //необходимо уменьшить счетчик количества читателей, которые
//прочитать еще не успели
    WaitForSingleObject(changeCountEvent,INFINITE);

    (*((int *) lpFileMapForReaders)--;
    printf("readcount= %d\n",
        (*((int *) lpFileMapForReaders)));
    if((*((int *) lpFileMapForReaders))==0)
    {
        ResetEvent(canReadEvent); //если последнее
//чтение, запрещаем читать
        SetEvent(allReadEvent); //открываем границу
    }
    SetEvent(changeCountEvent);
    break;
default:
    printf("error with func WaitForMultipleObjects in
readerHandle\n");
    printf("getlasterror= %d\n",GetLastError());
    getchar();
    ExitProcess(1001);
    break;
}
}

exit:
//закрываем HANDLE объектов синхронизации
CloseHandle(canReadEvent);
CloseHandle(canWriteEvent);
CloseHandle(allReadEvent);
CloseHandle(changeCountEvent);
CloseHandle(exitEvent);

//закрываем общий ресурс
UnmapViewOfFile(lpFileMapForReaders);
CloseHandle(hFileMapping); //закрываем объект "отображаемый файл"

printf("all is done\n");

```

```
getchar();  
return 0;  
}
```

Выполнение программы

Главный поток ждет завершения всех процессов-читателей (вернее их главных потоков). Т.к. потоки-читатели, принадлежащие разным процессам, имеют каждый свою консоль, то после окончания работы они сразу не завершаются, а ждут ввода любой клавиши. Каждый поток-читатель успевает читать каждое сообщение.

Альтернативным решением задачи читатели-писатели может быть использование функции **PulseEvent** (освобождение и сразу блокировка события (чтобы проснулись потоки, ожидающие событие)), особенности ее работы необходимо учитывать при разработке алгоритма. Данная функция осуществляет пробуждение *только тех* потоков, *которые* на момент вызова *уже ожидали* освобождения события (объект-событие с автосбросом). Если какой-либо поток не успел вызвать функцию wait, и была вызвана функция PulseEvent, то поток так и останется в ожидании. Таким образом, использование функции PulseEvent не является безопасным. Кроме того, данная функция при действии на событие с ручным сбросом пробуждает *все потоки, которые* на момент вызова *ожидали* освобождения события (т.к. находились в вызове wait). Если какой-либо из потоков не успеет вызвать wait, то он не будет разбужен, т.к. PulseEvent открывает и сразу закрывает событие.

В приведенном решении вместо использования данной функции вводятся 2 события с ручным сбросом (*allReadEvent* и *canReadEvent*), освобождение и «захват» которых осуществляют потоки-читатели.

Недостатком представленной реализации для потоков разных процессов является то, что *счетчики*, по которым синхронизируются потоки-читатели, *пишутся в общую память*. Поэтому потоки-читатели имеют полный доступ к памяти (и на чтение, и на запись), что противоречит исходному заданию, ухудшает характеристики обмена и не является безопасным. Потоки не должны модифицировать память.

Данную проблему можно решить *несколькими способами*:

- 1) Счетчики должны храниться в процессе потока-писателя. Потоки-читатели вместо декремента счетчика производят освобождение событий, по которым поток-писатель декрементирует счетчик.

- 2) Можно ввести поток-координатор, который осуществлял бы поэтапное выполнение задания. В итоге, такая реализация потребовала бы добавления еще 3-х событий: «1» - чтобы сигнализировать в каждый момент времени мог только 1 поток-читатель (т.е. только 1 читатель имел бы доступ к коду), «2» - событие-сигнал для потока-писателя на декремент счетчика, «3» - для разрешения потоку освобождения события «1»;
- 3) Вместо счетчиков использовать 1 семафор. До начала «критической секции» семафору присваивается значение, равное количеству потоков-читателей. При этом поток-писатель ожидает обнуления семафора. Потоки-читатели, проходя критическую секцию, декрементируют семафор.

Выполнить самостоятельно:

1. Привести собственные результаты выполнения предложенных программ и их анализ.
2. Модифицировать предложенное решение таким образом, чтобы «читатели» не имели доступа к памяти по записи.
3. Предложить более рациональное решение задачи «читатели-писатель», используя другие средства синхронизации или их сочетание. Объяснить и подтвердить экспериментально улучшение характеристик взаимодействия.
4. Разработать клиент-серверное приложение для полной задачи «читатели-писатели» с собственной системой ограничений на доступ каждого «читателя» к информации.
5. Разработать программу «читатели-писатели» для сетевого функционирования. Для этого выбрать подходящие средства IPC и синхронизации
6. Предложить программное решение задачи «производители-потребители». Разница с предыдущей задачей – возможность модификации считываемых данных.
7. Решить задачу «обедающие философы», обосновать выбранные средства синхронизации.

Заключение к разделам по Windows

Операционная система Windows предоставляет множество инструментов для обеспечения синхронизации по доступу к ресурсу внутри одного процесса. Некоторые из инструментов могут применяться по отдельности (семафоры, мьютексы), а некоторые только в паре с другими (условные переменные).

Все реализации выше (за исключением использования условных переменных) производят периодический опрос состояния очереди. Т.е. они захватывают объект-синхронизации, проверяют, могут ли использовать очередь. Если не могут, то просто освобождают и опять становятся в очередь к объекту. Причиной этому является то, что для синхронизации использовался только 1 объект (а не 3, как с условными переменными). С условными переменными мы с помощью передаваемых сигналов пробуждаем потоки, когда это действительно необходимо (когда в очереди есть данные, запускаются потоки – читатели, если в очереди есть место, то – потоки-писатели).

В общем случае все средства синхронизации отличаются как по использованию (только между потоками внутри процесса или еще и между процессами), так и по быстродействию (в работе измерения не проводились). Самым быстрым средством считается критическая секция, т.к. она выполняется в режиме задачи и максимально упрощена.

Мьютексы являются аналогом семафора с двумя возможными значениями. Реализуют взаимное исключение, т.е. одновременно доступ к мьютексу имеет только 1 поток. Мьютексы так же могут быть использованы для синхронизации как потоков, так и процессов.

Семафор позволяет захватить себя нескольким потокам, после чего захват будет невозможен, пока один из ранее захвативших семафор потоков не освободит его. Семафоры применяются для ограничения количества потоков, одновременно работающих с ресурсом (ресурсами).

В Windows семафоры и мьютексы могут иметь имя, через которое другие процессы могут получить доступ к объектам. В отличие от синхронизации потоков, принадлежащих одному процессу, при межпроцессной синхронизации в каждом процессе должен быть получен свой дескриптор к одному и тому же объекту ядра.

Критическая секция помогает выделить участок кода, где поток получает доступ к разделяемому ресурсу, и предотвратить одновременное использование ресурса. К критической секции одновременно имеет доступ только 1 поток. Данное средство **не может быть применено для синхронизации процессов**. Критическая секция **не является объектом ядра**, и использовать функции семейства Wait() не представляется возможным.

События используются для уведомления ожидающих потоков о наступлении какого-либо события. Существует два вида событий - с ручным и автоматическим сбросом. События с ручным сбросом используются для уведомления сразу нескольких потоков. При использовании события с

автоматическим сбросом уведомление получит и продолжит свое выполнение только один ожидающий поток, остальные будут ожидать дальше.

Практически все средства синхронизации (кроме критических секций) используются вместе с **ожидающими функциями**, которые возвращают управление, если все или часть переданных им объектов свободны.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *А.М. Робачевский* Операционная система UNIX: учебник / А.М. Робачевский [и др.]. – 2-е изд., перераб. и доп. – СПб. : Изд-во БХВ-Петербург. 2010. – 648 с.
2. *А. Таненбаум* Современные операционные системы / А. Таненбаум, Х. Бос.– 4-е изд., перераб. и доп.– СПб.: Изд-во Питер. 2015. – 1120 с.
3. *М. Руссинович, Д. Соломон* Внутреннее устройство Microsoft Windows / М. Руссинович, Д. Соломон – 6-е изд., перераб. и доп. – СПб. : Изд-во Питер. 2013. – 800 с.
4. *Р. Лав* Ядро Linux. Описание процесса разработки – М.: Изд-во Вильямс. 2014. – 496 с.
5. *В. Олифер, Н. Олифер* Сетевые операционные системы: учебник для ВУЗОВ / В. Олифер, Н. Олифер. – 2-е изд., перераб. и доп. – СПб. : Изд-во Питер. 2009. – 672 с.
6. *У. Стивенс* UNIX: Взаимодействие процессов : учебник / У. Стивенс. – СПб. : Питер, 2002. – 398 с.
7. *С. Максвелл* «Ядро Linux в комментариях»– Киев : Изд-во Диасофт. 2000. – 488 с.
8. *Г. Р. Эндрюс* Основы многопоточного, параллельного и распределенного программирования – СПб. : Изд-во Вильямс. 2003. – 512 с.
9. *Е.В. Душутина* Системное программное обеспечение. Практические вопросы разработки системных приложений: учеб. пособие – СПб. : Изд-во Политехн. ун-та, 2016. – 156 с. (ISBN 978-5-74225014-2)