Министерство образования и науки Российской Федерации

САНКТ-ПЕТЕРБУРГСКИЙ
ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО

В. Э. Шмаков М. В. Хлудова

ОТКРЫТЫЕ СИСТЕМЫ И LINUX-ТЕХНОЛОГИИ

Учебное пособие

ИЗДАТЕЛЬСТВО
ПОЛИТЕХНИЧЕСКОГО УНИВЕРСИТЕТА

Санкт-Петербург
2018

ББК 32.972.5
Ш71

Р е ц е н з е н т – кандидат технических наук, доцент кафедры «Компьютерные системы и программмные технологии» Института компьютерных наук и технологий СПбПУ Петра Великого *Н. В. Богач*

Шмаков В. Э. **Открытые системы и Linux-технологии** : учеб. пособие / В. Э. Шмаков, М. В. Хлудова. – СПб. : Изд-во Политехн. ун-та, 2018. – 58 с.

Учебное пособие содержит теоретические сведения и набор заданий для проведения лабораторных работ по изучению открытых операционных систем семейства Linux.

В первой части внимание уделяется изучению файловой системы, прав доступа, принципов запуска и завершения процессов в операционной среде, их синхронизации и взаимодействию. Рассматриваются такие технологии, как конвейеры, очереди сообщений, семафоры и разделяемая память. Изучаются сетевые возможности Linux, а также основы сетевой безопасности, средства анализа и фильтрации сетевого трафика, построения программных сетевых экранов.

Во второй части основное внимание уделяется изучению возможностей подсистемы управления потоками. Рассматриваются процедуры создания и завершения потоков, механизмы синхронизации доступа потоков к разделяемым ресурсам процесса, а также соотношение потоков и процессов.

Предназначено для студентов и преподавателей, проводящих занятия по дисциплинам «Открытые системы и Linux-технологии», «Администрирование в информационных системах», «Операционные системы», «Основы сетевой безопасности» и т. д.

Ил. 9. Библиогр.: 7 назв.

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

ISBN 978-5-7422-6178-0
doi:10.18720/spbpu/2/id18-15

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Операционная система (ОС) *Linux* является широко известным и типичным представителем класса открытых систем, и ей отводится все большая часть в учебных планах университетов и колледжей. Особо важным компонентом учебного процесса являются лабораторные занятия, где студенты имеют возможность получить практические навыки работы, администрирования и разработки ПО под *Linux*.

Тематика лабораторных работ из первой части (№ 1–15) тесно переплетается с содержанием основного курса лекций, а примеры многих программ рассматриваются в лекциях, и их исходные файлы имеют такие же имена, как указаны в заданиях лабораторных работ первой части. Поэтому изложение теоретического материала в описании лабораторных работ первой части минимизировано, чтобы не дублировать лекции. В качестве основной литературы по первой части рекомендуется, в первую очередь, использовать конспект лекций и, кроме того, в изданиях [2–4, 7] можно найти много полезной информации.

Описание лабораторных работ из второй части (№ 16–18) учебного пособия сопровождается еще и достаточно подробным изложением теоретического материала, поскольку вторую часть следует рассматривать как развитие первой, а тематика второй части носит факультативный характер в силу повышенной сложности. В дополнение к приведенному в описании работ теоретическому материалу рекомендуются также издания [5, 6].

Ресурсы Интернета в области открытых систем и Linux-технологий поистине неисчерпаемы. В данной работе приводятся местами конкретные ссылки, но это не значит, что следует ограничиваться лишь ими. Каждый сможет выбрать себе в поисковиках такие ресурсы, которые будут в наибольшей степени отвечать его индивидуальным запросам и наклонностям в манере представления материала.

Часть 1

ОСНОВЫ РАБОТЫ И СЕТЕВЫЕ ВОЗМОЖНОСТИ ОС *LINUX*

Лабораторная работа № 1
БАЗОВЫЕ КОМАНДЫ ОС

Цель работы. Освоение минимального набора базовых команд операционной системы *Linux*, знакомство с файловой системой, особенностями прав доступа, получение первичных навыков работы под *Linux*.

Последовательность выполнения работы:

1. Войдите в систему под логином вашей учебной группы, получив необходимый пароль у преподавателя.

2. Запустите терминал нажатием комбинации клавиш *Ctrl + Alt + t* .

3. Выполните на терминале команды *shell* , рассмотренные в материалах лекций. Такие, как *pwd* , *who* , *ls* , *cd* , *mkdir* , *rm* , *chmod* . Полное описание синтаксиса и семантики этих и любых других команд можно увидеть в системе помощи ОС *Linux*, вызываемой с терминала в виде *man < интересующая вас команда >*, или запускайте веб-браузер и используйте всю информационную мощь Интернета.

4. Проанализируйте результаты выполнения команд. Наиболее значимые скриншоты (снимаются нажатием клавиш *Alt + Prnt Scrn*) поместите в отчет.

5. Создайте дерево каталогов глубиной вложения до трех уровней, а в самих каталогах создайте текстовые файлы. Примените различные способы создания новых файлов.

6. Запустите с терминала *Midnight Commander* вводом команды *mc* и ознакомьтесь с его основными возможностями по работе с файловой системой. Наполните созданные на предыдущем шаге файлы каким-либо содержанием. Для этого можно использовать

любой редактор, от *vi* , встроенного в ОС, до графического редактора *gedit* , вызываемого из графической оболочки ОС.

7. Выполните на терминале вторую серию команд *cat* , *cp* ,*find* , *link* , *chmod* , рассмотренных в лекциях. Для манипуляций с помощью этих команд используйте текстовые файлы, созданные и наполненные на предыдущем шаге.

8. Попытайтесь создать на своем дереве какой-нибудь каталог с правами доступа, аналогичными каталогу *darkroom* , рассмотренному в лекциях.

Лабораторная работа № 2
ЗАПУСК И ЗАВЕРШЕНИЕ ПРОЦЕССОВ

Цель работы. Знакомство с характерной для *Linux* схемой порождения и завершения процессов, с отношениями типа потомок – родитель, со способами передачи информации о событии завершения процесса.

В данной и в последующих работах понадобится компилировать исходные файлы с помощью строчного компилятора *g++* . В простейших случаях одномодульных проектов (до работы № 16) применяется команда *g++ < имя.cpp файла >* , исполняемый файл генерируется только в случае отсутствия ошибок компиляции и по умолчанию именуется *a.out.* Запускается исполняемый файл с терминала с указанием полного пути (например, */a.out*, если он в текущем каталоге) или из *Midnight Commander-a.* Сведения о полной функциональности *g++* можно почерпнуть в справке *man* .

Последовательность выполнения работы:

1. Войдите в систему и скопируйте в свой HOME-каталог с разделяемого ресурса набор исходных файлов для второй лабораторной работы.

2. Скомпилируйте и выполните примеры программ *forkdemo. cpp* , *tinymenu.cpp* , *tinyexit.cpp* , *procgroup.cpp* , *wait_parent.cpp* . Процесс *wait_parent* при исполнении запускает процесс *wait_child* . Программа *wait_child.cpp* компилируется с опцией: *g++ wait_ child.cpp-o wait_child* . Пояснения к данным программам можно

найти в тексте лекций. При необходимости конвертации текстовых файлов из формата *DOS* в *Linux* и наоборот используйте команды *dos2unix* и *unix2dos* .

3. Модифицируйте программу *forkdemo.cpp* (или создайте собственную), так чтобы ввод/вывод на терминал отсутствовал, а при проходе по циклу была временная задержка, например *sleep (7)*. Запустите эту программу в фоновом режиме (*background*), введя при запуске символ *&* после пробела и зафиксировав значение *PID*, назначенное системой фоновому процессу при запуске. Выполните на терминале команды *ps* , *top* , *uptime* , *pstre* . Снимите свой фоновый процесс командой *kill* с соответствующими параметрами. Скриншоты вместе с пояснениями к выполнению процессов и команд, а также исходные тексты программ, составленных вами самостоятельно, приведите в отчете.

4. Исследуйте, что произойдет, если процесс-потомок сменит текущий каталог, будет ли изменен текущий каталог для родителя? Создайте программу, подтверждающую ответ и приведите в отчете.

5. Проиллюстрируйте как процесс-родитель и процесс-потомок разделяют один и тот же дескриптор и смещение текстового файла. Для этого составьте программу, в которой процесс-родитель должен открывать текстовый файл и запускать потомка. Потомок должен читать порцию данных из открытого файла и выводить на консоль. По завершению потомка родитель должен читать из того же файла и выводить результат на консоль. Можете использовать вызов *sleep()* для синхронизации доступа родителя и потомка к файлу.

Лабораторная работа № 3
ПРОГРАММНЫЕ КАНАЛЫ

Цель работы. Изучение конвейеров (*pipes*, программных каналов), как простейшего средства коммуникации запущенных процессов. Исследование различных способов организации каналов и их сопоставление.

Последовательность выполнения работы:

1. Войдите в систему и скопируйте с разделяемого ресурса в свой HOME-каталог набор исходных файлов для третьего занятия.

2. Скомпилируйте и выполните программу *whosortpipe.cpp* . Сопоставьте результат выполнения программы с выполнением этих же двух команд из *shell* в конвейерном режиме (|). Не забывайте приводить в отчете анализ результатов работы этой программы (как и всех последующих) с соответствующими скриншотами.

3. Программу *cmdpipe.cpp* запускайте после компиляции, задавая ей при стартах в качестве параметров командной строки пары команд *shell* для конвейеризации (*who* и *sort* ; *last* и *sort* ; *last* и *more* ; *pstree* и *more*). Сопоставьте результаты запусков программы с выполнением тех же пар команд из *shell* в конвейерном режиме. Можно ли с помощью вызова *popen()* создать программу, организующую конвейер из трех команд *shell* , передаваемых ей в качестве параметров командной строки при запуске? Если да, то создайте такую программу, если нет, дайте обоснованный ответ, почему нельзя.

4. Напишите программу (например, на основе вызовов *pipe()*), воспринимающую варьируемое количество команд, передаваемых ей при запуске в качестве параметров. Каждая последующая команда должна быть соединена с предыдущей с помощью конвейера. Так, при запуске программы

$ *./a.out last sort more*

должны выполняться действия, эквивалентные запуску команд из *shell* :

$ *last* | *sort* | *more.*

5. Разберите и выполните пример клиент-серверного взаимодействия, организованного на конвейерах различного типа. Исходный текст примера содержится в файлах *pipe_server.cpp* , *pipe_client.cpp* и *pipe_local.h* и разобран в материалах лекций. Сервер запускается в фоновом режиме. Проанализируйте результаты

функционирования данной системы и ее недостатки. Программа-сервер этого примера исполняет каждый командный запрос поочередно. Если какой-либо запрос потребует много времени, все остальные клиентские процессы будут ожидать обслуживания.

6. Модифицируйте программу *pipe_server.cpp* так, чтобы при получении нового сообщения от очередного клиента сервер порождал очередной дочерний процесс для выполнения задачи обслуживания данного запроса (выполнения переданной от клиента команды и переправки клиенту результата).

Лабораторная работа № 4
КОМАНДНЫЕ ФАЙЛЫ. ПЕРЕМЕННЫЕ ОКРУЖЕНИЯ

Цель работы. Знакомство с важным атрибутом любой операционной системы – переменными среды (или переменными окружения) и с возможностями их использования в *Linux*. Освоение языка для составления командных сценариев и написание набора полезных для системного администрирования скриптов.

Последовательность выполнения работы:

1. Создайте несколько символьных переменных среды (переменных окружения). Составьте командный файл (сценарий *bash)*, выводящий на консоль значения этих переменных. Выполните операцию конкатенации (склеивания) значений переменных и выведите полученный результат на консоль. Выделите из конкатенированной переменной среды подстроку и выведите ее на консоль. Замените выделенную подстроку на какое-либо другое значение и выведите измененное значение переменной среды на консоль.

2. Создайте несколько переменных среды в интерпретации, как числовые переменные. В новом командном файле выполните с этими числовыми переменными все допустимые арифметические операции, выводя на консоль результаты операций и соответствующие комментарии.

3. Создайте командный файл (основной), выдающий при старте сообщение и затем вызывающий другой командный файл (его

имя задается при старте основного файла в качестве параметра командной строки), который выдает свое сообщение и приостанавливается до нажатия любой клавиши. При возврате управления в вызывающий (основной) файл из него должно выдаваться еще одно сообщение, подтверждающее возврат.

4. Составьте командный файл, выводящий на экран различия содержимого двух каталогов, имена которых передаются в качестве параметров. Отличия искать в именах файлов, их размерах и атрибутах.

5. Разработайте командный файл сценария для поиска текстовых файлов, содержащих заданную последовательность символов. Эта последовательность передается при запуске в качестве первого параметра командной строки. В качестве второго параметра передается имя файла результатов, который должен быть создан в сценарии для записи в него имен найденных текстовых файлов и номеров их строк, в которых содержится заданная последовательность символов.

6. Разработайте командный файл сценария для поиска текстовых файлов, содержащих заданную последовательность символов (эта последовательность передается при запуске в качестве первого параметра командной строки). В качестве второго параметра передается имя файла результатов, который должен быть создан в сценарии для записи в него имен найденных текстовых файлов и номеров их строк, в которых содержится заданная последовательность.

7. Создайте командный файл, который синхронизирует содержимое заданного каталога с эталонным. После запуска и отработки командного файла в заданном каталоге должен оказаться тот же набор файлов, что и в эталонном (если файла нет – он копируется из эталонного каталога, если найдется файл, которого нет в эталонном, – удаляется). Если файл с некоторым именем есть и в заданном и в эталонном каталогах, то он перезаписывается только в том случае, если в эталонном имеется более новая версия файла. Имена обоих каталогов должны при запуске передаваться командному файлу параметрами командной строки.

Лабораторная работа № 5

УЧЕТНЫЕ ЗАПИСИ. ФОНОВЫЙ И ДИАЛОГОВЫЙ РЕЖИМЫ ИСПОЛНЕНИЯ ПРОЦЕССОВ

Цель работы. Манипуляции с правами доступа при создании в системе учетных записей и исследование влияния прав на файловые операции. Изучение специфик фонового (*background*) и диалогового (*foreground*) режимов исполнения процессов и способов переключений между этими режимами.

Последовательность выполнения работы:

1. Создайте учетные записи для нескольких пользователей (не задавая им прав администратора) и объедините их в две группы. Заходя в систему под разными аккаунтами, создайте в соответствующих домашних каталогах файлы, варьируя при этом права доступа для пользователя, для группы, для всех. Убедитесь, что права доступа разделяются в соответствии с тем, как это задано. Проведите операцию слияния файлов с различными правами доступа и проверьте, какие при этом получаются права у результирующего файла.

2. Запустите в фоновом (*background*) режиме командный файл (процесс), выдающий в цикле с некоторой задержкой сообщение на консоль. Запустите другой командный файл (процесс), требующий диалога, в обычном режиме (*foreground*). Убедитесь в том, что вывод этих двух процессов на консоль перемежается. Остановите фоновый процесс сигналом *kill* . Запустите его снова, организовав предварительно перенаправление его вывода в файл. Убедитесь, что теперь вывод двух процессов разделен.

3. Доработайте предыдущее задание так, чтобы показать возможность перевода фонового процесса в диалоговый режим (*foreground*) для выполнения операции ввода с клавиатуры и затем возврата его обратно в фоновый (*background*) режим (команды *fg* , *bg* , *jobs*). Продемонстрируйте возможность оставления фонового процесса на исполнение после завершения пользовательского сеанса работы в ОС.

4. Разработайте командный файл для выполнения архивации каталога через определенные интервалы времени. Запустите командный файл в режиме *background* . Имя архивируемого каталога, местоположение архива и время (период) архивации передаются при запуске командного файла в виде параметров командной строки.

Лабораторная работа № 6
ГЕНЕРАЦИЯ И ОБРАБОТКА СИГНАЛОВ

Цель работы. Освоение простейшего средства управления процессами, позволяющего процессам передавать информацию о каких-либо событиях, отрабатывать реакции на различные события и взаимодействовать друг с другом.

Последовательность выполнения работы:

1. Войдите в систему и скопируйте с разделяемого ресурса в свой HOME-каталог набор исходных файлов для шестого занятия.

2. Программа *sigint.cpp* осуществляет ввод символов со стандартного ввода. Скомпилируйте и запустите программу и отправьте ей сигналы *SIGINT* (нажатием *Ctrl-C*) и *SIGQUIT* (нажатием *Ctrl-*). Проанализируйте результаты.

3. Запустите программу *signal_catch.cpp* , выполняющую вывод на консоль. Отправьте процессу сигналы *SIGINT* и *SIGQUIT* , а также *SIGSTOP* (нажатием *Ctrl-Z*) и *SIGCONT* (нажатием *Ctrl-Q*) . Проанализируйте поведение процесса и вывод на консоль, а также сравните с программой из предыдущего пункта.

4. Скомпилируйте и запустите программу *sigusr.cpp* . Программа выводит на консоль значение ее *PID* и зацикливается, ожидая получения сигнала. Запустите второй терминал и, отправляя с него командой *kill* различные сигналы, в том числе и *SIGUSR1* , проанализируйте реакцию на них.

5. Составьте программу, запускающую процесс-потомок. Процесс-родитель и процесс-потомок должны генерировать (можно

случайным образом) и отправлять друг другу сигналы (например, *SIGUSR1*, *SIGUSR2*). Каждый из процессов должен выводить на консоль информацию об отправленном и о полученном сигналах.

6. Для организации обработчиков сигналов предпочтительно использовать системный вызов *sigaction()* и соответствующую структуру данных. Обеспечьте корректное завершение процессов.

7. Модифицируйте программу *занятия 3* (файлы *pipe_server.cpp* , *pipe_client.cpp* и *pipe_local.h*), сделав ее более стабильной в работе. В числе недостатков, которые желательно устранить, можно указать:

– если клиентский процесс завершается по получению сигнала *SIGINT* (*Ctrl+C)*, то *private FIFO* не удаляется из системы (исправляется посредством организации перехвата сигнала с выполнением необходимых действий);

– клиентский процесс при его инициализации может обрушиться, если сервер окажется недоступен (исправляется путем попытки запуска сервера из клиента, если сервер не активен).

Лабораторная работа № 7
СЕМАФОРЫ И СИНХРОНИЗАЦИЯ

Цель работы. Освоение семафоров (semaphores) как эффективных средств синхронизации доступа процессов к разделяемым ресурсам операционной системы, а также синхронизации доступа потоков (в части 2) к разделяемым ресурсам процесса.

Последовательность выполнения работы:

1. Войдите в систему и скопируйте с разделяемого ресурса в свой HOME-каталог набор исходных файлов для седьмого занятия.

2. Скомпилируйте и выполните программу *gener_sem.cpp* , иллюстрирующую создание наборов с семафорами или получение доступа к ним. Запустите программу несколько раз и после каждого ее завершения выполните команду *ipcs -s* . Поясните зависимость процедуры создания семафоров от используемых в вызове *semget()* флагов.

3. Удалите созданные на предыдущем шаге семафоры с помощью команды *ipcrm* с соответствующей опцией и значением *id* семафора или ключа.

4. Скомпилируйте s*emdemo.cpp* , демонстрирующую организацию разделения доступа к общему ресурсу между несколькими процессами с помощью технологии семафоров. Запустите сразу несколько процессов на разных терминалах и проанализируйте их взаимодействие и соблюдение очередности в попытках получения общего ресурса.

5. Скомпилируйте программу *semrm.cpp* и произведите с ее помощью удаление созданного на предыдущем шаге семафора. Поясните, почему данная программа удаляет только те семафоры, которые были созданы при выполнении программы *semdemo.cpp* .

6. Попробуйте удалить семафор с помощью запуска *semrm.cpp* во время исполнения *semdemo.cpp* и проанализируйте ситуацию.

7. Попытайтесь улучшить программу *semdemo.cpp* , например, предоставив процессу возможность после освобождения ресурса становиться снова в очередь на повторное его занятие (а не завершаться), организовав при этом завершение процесса по вводу какого-либо символа.

8. Составьте программу, позволяющую мониторить количество процессов (типа *semdemo*), находящихся в состоянии ожидания освобождения ресурса (*Trying to lock...*) в каждый момент времени. Программа строится на основе вызова *semctl()* с соответствующими параметрами и запускается на отдельном терминале.

Лабораторная работа № 8
ОБМЕН ЧЕРЕЗ ОЧЕРЕДИ СООБЩЕНИЙ

Цель работы. Знакомство с возможностями очередей сообщений (*Message Queues*) – мощного и гибкого средства межпроцессного взаимодействия в ОС *Linux.*

Последовательность выполнения работы:

1. Войдите в систему и скопируйте с разделяемого ресурса в свой HOME-каталог набор исходных файлов для восьмого занятия.

2. Скомпилируйте и выполните программу *gener_mq.cpp* , создающую несколько очередей сообщений. После завершения программы выполните команду *ipcs* и поясните отличие результата от того, что был при вызове подобной команды из программы.

3. Скомпилируйте программы *sender.cpp* и *receiver.cpp* , задав соответствующим исполняемым файлам разные имена (*g++ < < имя .cpp файла > -o < имя .out файла >*). Запустите процессы на разных терминалах и передайте текстовые сообщения от процесса *sender* процессу *receiver.* Проанализируйте, что происходит с ресурсом *Message Queue* после завершения каждого из процессов (командой *ipcs*). При этом выполните различные виды завершения отправкой сигналов *SIGQUIT* и *SIGINT* (нажатием *Ctrl-C*).

4. Ответьте на вопрос: что происходит, если процесс *receiver* запускается уже после того, как процесс *sender* отправил в очередь одно или множество сообщений?

5. Запустите несколько процессов *receiver* на различных терминалах и, отправляя сообщения процессом *sender* , проанализируйте ситуацию.

6. Модифицируйте программы *sender.cpp* и *receiver.cpp* так, чтобы организовать отправку сообщений двух типов через одну и ту же очередь для двух различных процессов получателей. Для этого необходимо управлять параметром в поле *mtype* структуры *my_msgbuf* на передающей стороне и параметром *msgtyp* в системном вызове *msgrcv()* на приемной стороне.

Лабораторная работа № 9
РАБОТА С РАЗДЕЛЯЕМОЙ ПАМЯТЬЮ

Цель работы. Использование для обмена данными разделяемой памяти (*shared memory*) – самого быстрого средства межпроцессного взаимодействия в *Linux*.

Последовательность выполнения работы:

1. Войдите в систему и скопируйте с разделяемого ресурса в свой HOME-каталог набор исходных файлов для девятого занятия.

2. Скомпилируйте и выполните программу *gener_shm.cpp* демонстрирующую создание сегментов разделяемой памяти. Запустите программу несколько раз и после каждого ее завершения выполните команду *ipcs -m* . Поясните зависимость процедуры создания сегментов разделяемой памяти от используемых в вызове *shmget()* флагов.

3. Удалите созданные на предыдущем шаге сегменты разделяемой памяти с помощью команды *ipcrm* с соответствующей опцией и значением *id* сегмента или ключа.

4. Скомпилируйте s*hmdemo.cpp* , осуществляющую операции записи в разделяемую память без разделения доступа к этому общему ресурсу. Символы, записываемые в общую память, передаются в качестве параметра командной строки при запуске процесса *shmdemo* . Запуск этого процесса без параметров приводит к выводу на консоль текущего содержимого сегмента общей памяти.

5. Запустите несколько раз процессы типа s*hmdemo* с различными значениями параметров и проиллюстрируйте возможности чтения и записи в сегмент общей памяти независимо исполняемыми процессами. Затем удалите сегмент памяти командой *ipcrm* .

6. Скомпилируйте и выполните программу *attach_shm.cpp* , иллюстрирующую передачу символьной информации между двумя процессами (родственными) через сегмент общей памяти с модификацией этой информации. Проанализируйте значения выводимой информации о границах сегментов в системной памяти. За счет чего после завершения данной программы сегмент общей памяти уже не присутствует в системе?

7. Составьте программу, создающую три разделяемых сегмента памяти размером 1023 байта каждый. Укажите в вызове *shmat()* параметр *shmaddr = 0* при привязке сегментов. Разместит ли система сегменты в последовательных участках? Позволит ли система ссылку или изменение 1024-го байта любого из этих участков?

Лабораторная работа № 10

СОЗДАНИЕ СОЕДИНЕНИЙ НА СОКЕТАХ

Цель работы. Освоение набора системных вызовов для создания сокетных соединений различных типов для обмена данными по сети.

Последовательность выполнения работы:

1. Войдите в систему и скопируйте с разделяемого ресурса в свой HOME-каталог набор исходных файлов для десятого занятия.

2. Скомпилируйте и выполните программу *socketpair.cpp* , иллюстрирующую создание простейшего вида сокета и обмен данными двух родственных процессов. Проанализируйте вывод на консоль. Существует ли зависимость обмена от различных соотношений величин временных задержек (в вызовах *sleep()*) в процессе-родителе и в процессе-потомке?

3. Скомпилируйте программы *echo_server.cpp* и *echo_client.cpp* , задавая им при компиляции разные имена (размещаем файлы в одном каталоге). Запустите программы сервера и клиента на разных терминалах. Введите символьную информацию в окне клиента и проанализируйте вывод. Какой разновидности принадлежат сокеты, используемые в данном примере клиент-серверного взаимодействия? С чем связано создание специального файла в текущем каталоге во время исполнения программ?

4. Скомпилируйте с разными именами программы *sock_c_i_srv*. *cpp* и *sock_c_i_clt.cpp* (в них используется общий *include* файл *local_c_i.h*). Запустите программы сервера и клиента на разных терминалах. При запуске клиента указывайте в качестве параметра командной строки имя хоста *localhost* . Введите символьную информацию в окне клиента и поясните вывод. Какой разновидности принадлежат сокеты, используемые в данном примере клиент-серверного взаимодействия?

5. Модифицируйте программу *echo_server.cpp* так, чтобы при ответе на запросы клиента что-либо выводилось в окне сервера. Испытайте работу эхо-сервера при одновременной работе с несколькими клиентами.

Лабораторная работа № 11

ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ ПО СЕТИ

Цель работы. Создание клиент-серверных приложений, взаимодействующих друг с другом по сети на основе технологии соединения на сокетах.

Последовательность выполнения работы:

1. Войдите в систему и скопируйте с разделяемого ресурса в свой HOME-каталог набор исходных файлов для одиннадцатого занятия.

2. Скомпилируйте и запустите программу *server_game.cpp* , иллюстрирующую обмен данными с клиентскими приложениями по итеративной схеме.

3. Запустите другой терминал и проверьте с него наличие в системе созданного сервером сокета и то, что он находится в состоянии *LISTEN*. Для этого выполните команду *netstat -a | grep 1066* . Проанализируйте вывод данной команды и объясните ее смысл.

4. Запустите в качестве клиентского процесса утилиту *telnet* с параметрами: *telnet localhost 1066* . При организации коммуникации по сети на разных компьютерах вместо *localhost* при запуске клиента указывается *IP-адрес* компьютера, на котором был запущен сервер.

5. Диалог с сервером заключается в угадывании слова. Оно вводится по буквам с клиентского терминала. При этом сервер вместо неугаданных букв выдает символы ”-” , а также считает число оставшихся неудачных попыток (всего их предусмотрено 12).

6. Завершите серверное приложение с помощью сигнала *kill* , и затем определите командой *netstat -a | grep 1066* , когда исчезает из системы соединение на сокетах. Во время сеанса обмена также примените команду *netstat -a | grep 1066* , чтобы исследовать состояние соединения.

7. Проделайте все заново, но запускайте не одно клиентское приложение (в виде *telnet*), а несколько экземпляров с разных терминалов, и попытайтесь работать с них одновременно. Проанализируйте, как сервер будет обслуживать запросы в этом случае.

8. Модифицируйте программу *server_game.cpp* так, чтобы запросы от каждого из клиентов могли обслуживаться конкурентно (путем запуска для каждого нового соединения собственного нового процесса на сервере. Возможно также улучшить качество самой игровой функции *guess_word()* сервера. Проанализируйте, как обслуживаются запросы в случае конкурентной схемы работы сервера.

Лабораторная работа № 12
ФИЛЬТРАЦИЯ СЕТЕВОГО ТРАФИКА

Цель работы. Освоение технологии *iptables* , составление правил фильтрации сетевого трафика, выделение отдельных сетевых протоколов, *IP-адресов*, номеров портов. Создание простейших программных брандмауэров *(firewalls)*.

Последовательность выполнения работы:

1. Войдите в систему и определите *IP-адрес* Вашего компьютера. В дальнейшем также понадобится пароль администратора для вашего компьютера, пароль узнайте у преподавателя.

2. Просмотрите текущие правила, установленные в *iptables* . Результаты выполнения ваших команд, а также последствия применения вводимых вами правил протоколируйте в отчете о лабораторной работе. Никогда не сохраняйте вводимые правила командами типа *save* .

3. Введите правило, блокирующее весь входящий трафик на Ваш компьютер. Проверьте наличие введенного правила в системе.

4. Добавьте правила для фильтрации входящего трафика (с использованием целей *DROP* и *ACCEPT*) так, чтобы веб-трафик проходил, а остальной был блокирован. Просмотрите детализированный список правил и проверьте их действие.

5. Закройте вход *ICMP-пакетам*, отправляемым с какого-либо другого компьютера лаборатории. Создав соответствующее правило (испытайте цели *DROP* и *REJECT*), проверьте его действие утилитой *ping* .

6. Убедитесь, что при использовании цели *REJECT* , в отличие от *DROP* , выдается сообщение об ошибке на машину-отправитель.

7. Организуйте на Вашем компьютере форвардинг с какого-либо порта внешнего интерфейса на определенный порт другой машины лаборатории.

8. Проиллюстрируйте применение действия *MIRROR* таблицы *NAT*, используемое для защиты от сканирования портов.

Лабораторная работа № 13
ЖУРНАЛИРОВАНИЕ ТРАФИКА

Цель работы. Освоение различных вариантов журналирования (создания и ведения лог-файлов) передаваемых по сети пакетов с применением технологии *iptables* . Дифференцирование логов по разным событиям.

Последовательность выполнения работы:

1. Составьте правило логирования пингов и правило, разрешающее пинги:

iptables -A INPUT -p ICMP --icmp-type 8 -j LOG --log-prefix “Ping detected: ”

iptables -A INPUT -p ICMP --icmp-type 8 -j ACCEPT .

Правила с действием *LOG* изложены, например, на ресурсе http://www.opennet.ru/docs/RUS/iptables/#LOGTARGET .

2. Организуйте журналирование пакетов так, чтобы обеспечить запись в отдельный файл и не дублировать сообщения в системные логи.

Настройте ротацию логов *iptables* и убедитесь в ее правильности.

3. Попытайтесь создать правила для разных событий и при этом каждое событие направить в свой лог.

Лабораторная работа № 14

АНАЛИЗ СЕТЕВОГО ТРАФИКА

Цель работы. Получение навыков работы с мощным инструментом фильтрации трафика и обнаружения вторжений программой *Wireshark* .

Последовательность выполнения работы:

1. Установите и запустите в привилегированном режиме анализатор сетевого трафика *Wireshark* . О базовой функциональности снифера *Wireshark* можно узнать, например, из учебных роликов, выложенных на ресурсах:

http://www.youtube.com/watch?v=6X5TwvGXHP0;

http://www.youtube.com/watch?v=r0l_54thSYU;

http://www.youtube.com/watch?v=qs_DqMdlKHY .

2. Отфильтруйте трафик протокола *ICMP* (трафик порождается, например, утилитами *ping* , *traceroute*). Приведите в отчете подробный формат пакета, содержащего *ICMP-сообщение* с пояснением назначения каждого из полей.

Воспроизведите различные режимы работы утилит и приведите снятые снифером дампы пакетов с соответствующими этим режимам кодами сообщений или ошибок в полях пакетов.

3. Проанализируйте трафик *ARP* (протокола преобразования адресов). Поясните предназначение *ARP-таблиц* и приведите (с пояснениями) дампы *ARP-сообщений*, снятые снифером.

4. Установите на компьютере лаборатории *FTP-сервер*.

Выполните, по возможности, настройки, повышающие уровень защиты *FTP-сервера* (измените текст приветствия, организуйте отправку баннеров соединений, обезопасьте анонимный доступ), и проверьте работу настроек *FTP-сервера*, соединяясь с ним с клиентского приложения.

5. Продемонстрируйте уязвимость протокола FTP (в классической его версии имена и пароли пользователей передаются по незащищенным сетям в открытом виде) путем извлечения информации из пакетов с помощью анализатора трафика.

6. Сопоставьте защищенность протоколов удаленного доступа *Telnet* и *SSH* .

Действуйте по схеме, аналогичной демонстрации уязвимости протокола *FTP* .

Лабораторная работа № 15
ОГРАНИЧЕНИЕ ЧИСЛА СОЕДИНЕНИЙ

Цель работы. Применение сетевых технологий и утилит *Linux* для построения программных средств защиты от атак типа *DDOS* (отказа в обслуживании).

Последовательность выполнения работы:

1. Используйте технологию *iptables* и модуль *connlimit* для ограничения количества соединений к серверу с одного клиентского компьютера (защита от DDoS-атак). Например, разрешение не более трех соединений по *ssh* на одного клиента реализуется правилом:

iptables -A INPUT -p tcp --syn --dport 22 -m connlimit --connlimit-above 3 -j REJECT .

2. Продемонстрируйте возможность установления ограничения на количество одновременных соединений на примере *ssh* сервера или используйте собственный сервер из предыдущих разделов лабораторных работ по теме клиент-серверного взаимодействия на сокетах.

3. Проведите аудит локальной сети сканером *Nmap* . Не следует при этом сканировать порты серверов за пределами Вашего подразделения. Выбирайте в качестве объектов сканирования только компьютеры из своего сегмента локальной сети. Проверьте наличие открытых портов и протоколов, определите версию ОС на удаленном хосте, выполните по своему усмотрению 4–5 видов различного сканирования с помощью *Nmap* .

Информацию по функциональности *Nmap* можно найти, например, на ресурсах:

http://compress.ru/article.aspx?id=17371
https://www.youtube.com/watch?v=iUZ6nTMO8K0
https://www.youtube.com/watch?v=0xZqQDof-JA
https://www.youtube.com/watch?v=lXK5j2nRuv8
https://www.youtube.com/watch?v=kaJuZEW6D3I
https://www.youtube.com/watch?v=7CGvjshtkNk
https://www.youtube.com/watch?v=TyUtnOb-kS0 .

4. Организуйте с помощью утилиты *Netcat* взаимодействие процессов на разных хостах (наподобие *pipe* , но только на разных хостах). Используйте также утилиту *Cryptcat* и сравните защищенность соединений. Сведения об утилитах *Netcat* и *Cryptcat* можно узнать, например, из ресурсов:

http://handynotes.ru/2010/01/unix-utility-netcat.html
https://www.youtube.com/watch?v=oNwLy7JTJl8
https://www.youtube.com/watch?v=Ro7VDzOU32Q
https://www.youtube.com/watch?v=z04YgdWx4_Y .

Часть 2

СТАНДАРТ POSIX И ПОТОКИ В ОС LINUX

Лабораторная работа № 16
СОЗДАНИЕ, УПРАВЛЕНИЕ И УДАЛЕНИЕ ПОТОКОВ

1. Цель работы

Ознакомиться с подсистемой управления потоками в ОС *Linux* и основными программными средствами для создания, управления и удаления потоков.

2. Задание

Изучить основные программные средства управления потоками ОС *Linux*. Разработать приложения для многопоточных вычислений в соответствии с полученным вариантом.

3. Основные понятия

3.1. Поток

В настоящее время в мире существует несколько авторитетных сообществ [1], занимающихся разработкой стандартов «Открытых систем». Примером важных участников в этой области является созданные институтом IEEE рабочие группы и комитеты Portable Operating System Interface (POSIX). Стандарт POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995) определяет API для создания потоков, их синхронизации и планирования. Реализации данного API существуют для большого числа UNIX-подобных ОС (GNU/Linux, Solaris, FreeBSD, OpenBSD, NetBSD, OS X).

Библиотеки, реализующие этот стандарт и функции этого стандарта, обычно называются *Pthreads*, функции библиотеки имеют префикс *«pthread_»*.

Поток (*thread*) можно определить как ветвь процесса, включающую управляющую последовательность команд и использующую системные ресурсы этого процесса.

Каждый процесс имеет как минимум один поток, при этом самый первый поток, создаваемый при рождении нового процесса, принято называть начальным или главным потоком этого процесса.

Основное отличие процесса от потока заключается в способе использования системных ресурсов. Дочерний процесс практически независим от родительского, для него системой выделяется отдельное адресное пространство, и он на равных правах с родительским процессом «конкурирует» за процессорное время. При этом можно уничтожить родительский процесс, не затронув дочерний, который может выполняться и после завершения родительского процесса.

В отличие от дочернего процесса поток, порожденный данным процессом, полностью зависим от процесса, и завершение процесса влечет уничтожение всех созданных им потоков, поскольку происходит освобождение системных ресурсов, выделенных для этого процесса.

Еще одним отличием является невозможность изменить права доступа для потока, тогда как дочерний процесс в редких случаях (например, при смене пароля для входа в систему) может обладать правами доступа, отличными от прав родительского процесса.

Первая подсистема потоков в ОС *Linux* появилась в 1996 году и называлась *LinuxThreads*, ее автором является *Ксавье Лерой* (*Xavier Leroy*). Разработанная им библиотека *LinuxThreads* была попыткой организовать поддержку потоков в ОС *Linux*. Современная подсистема потоков ОС *Linux* стремится соответствовать требованиям стандартов *POSIX*, поэтому новые многопоточные приложения *Linux* должны компилироваться на новых *POSIX*-совместимых системах.

3.2. Программирование потоков в библиотеке Pthreads

В ОС *Linux* каждый поток на самом деле является самостоятельной единицей работы, и для того чтобы создать новый дополнительный поток, нужно создать новую единицу работы. Однако для создания дополнительных потоков используются процессы особого типа, разделяющие с главным процессом адресное пространство, файловые дескрипторы и обработчики сигналов. Для обозначения процессов этого типа применяется специальный термин – легковесные процессы (lightweight processes). Поскольку

для потоков не требуется создавать собственную копию адресного пространства (и многих других ресурсов) своего процесса-родителя, создание нового легковесного процесса требует значительно меньших затрат, чем создание полновесного дочернего процесса. Однако каждый новый легковесный процесс-поток имеет собственный стэк, указатель на выполняемую команду, состояние и сигнальную маску.

Спецификация международного стандарта *POSIX* 1003.1*c* требует, чтобы все потоки многопоточного приложения имели один идентификатор, однако в *Linux* у каждого процесса, в том числе и у процессов-потоков, есть свой уникальный идентификатор *PId*, который далее фигурирует как *tid*.

Pthreads определяет набор типов и функций на языке программирования Си, где заголовочный файл – *pthread.h* . Типы данных:

**pthread_t*: дескриптор потока,
**pthread_attr_t*: атрибуты потока.

Запуск и завершение потока. Потоки создаются функцией *pthread_create()*, имеющей сигнатуру:

int pthread_create (*pthread_t* tid*, *pthread_attr_t* attr*,
void(*function*)(*void*), *void* arg*) .

Данная функция определена в заголовочном файле < *pthread.h* > . Первый параметр этой функции представляет собой указатель на переменную типа *pthread_t* , которая служит идентификатором создаваемого потока. Второй параметр – указатель на переменную типа *pthread_attr_t* – используется для установки атрибутов потока. Третьим параметром функции *pthread_create()* должен быть адрес функции потока. Эта функция играет для потока ту же роль, что функция *main* для главной программы. Четвертый параметр функции *pthread_create()* имеет тип *void** . Этот параметр может использоваться для передачи значения в функцию потока.

Вскоре после вызова *pthread_create()* функция потока будет запущена на выполнение параллельно с другими потоками про-

граммы. Новый поток запускается не сразу после вызова *pthread_create()*, потому что перед тем как запустить новую функцию потока, нужно выполнить некоторые подготовительные действия, а поток-родитель при этом продолжает выполняться. Необходимо учитывать данное обстоятельство при разработке многопоточного приложения, в противном случае возможны серьезные ошибки при выполнении программы. Если при создании потока возникла ошибка, то функция *pthread_create()* возвращает ненулевое значение, соответствующее номеру ошибки.

Функция потока должна иметь сигнатуру вида

void func_name(void* arg)* .

Имя функции может быть произвольным. Аргумент *arg* является указателем, который передается в последнем параметре функции *pthread_create()*. Функция потока может вернуть значение, которое затем может быть обработано другим потоком или процессом. Функция, вызываемая из функции потока, должна обладать свойством реентерабельности (этим же свойством должны обладать рекурсивные функции). Реентерабельная функция – это функция, которая может быть вызвана повторно, в то время когда она уже вызвана. Такие функции используют локальные переменные (и локально выделенную память) в тех случаях, когда их нереентерабельные аналоги могут воспользоваться глобальными переменными.

Завершение функции потока происходит в случаях:

а) функция потока вызвала функцию *pthread_exit()* ;

б) функция потока достигла точки выхода;

в) поток был досрочно завершен другим потоком или процессом.

Функция *pthread_exit()* объявлена в заголовочном файле < *pthread.h* > и ее сигнатура имеет вид

void pthread_exit(*void *retval*) .

Аргументом функции является указатель на возвращаемый объект. Нельзя возвращать указатель на стековый (нединамический)

объект, объявленный в теле функции потока, либо на динамический объект, создаваемый и удаляемый в теле функции, так как после завершения потока все стековые объекты его функции удаляются. В итоге указатель будет содержать адрес памяти с неопределенным содержимым, что может привести к серьезной ошибке. В случае, если необходимо дождаться завершения неотсоединенного потока в теле родительского процесса, вызывается функция *pthread_join()* вида

int pthread_join(*pthread_t th, void** thread_return*) .

Первый аргумент *th* необходим для указания ожидаемого потока. Значение этого аргумента определяется в результате выполнения функции *pthread_create()*.

Вторым аргументом *thread_return* выступает указатель на аргумент функции *pthread_exit()* либо *NULL*, если поток ничего не возвращает.

Функция возвращает нуль в случае успешного присоединения, иначе она возвращает код ошибки: *EINVAL* – дескриптор *thread* указывает на нить, которая не может быть присоединена; *ESRCH* – нет нити с таким дескриптором *thread*; *EDEADLK* – обнаружен тупик (deadlock) или нить пытается завершить себя.

Невнимание к тому факту, что потоки по умолчанию создаются со значением атрибута «отсоединенность», равным *PTHREAD_ CREATE_JOINABLE* , и ресурсы таких потоков должны быть «утилизированы» т. е. возвращены родительскому процессу посредством вызова функции *pthread_join()* , может привести к ошибочной ситуации. Обратите внимание, что функция *pthread_exit()* не освобождает выделенную память.

Отсоединение нити позволяет освободить занятые ею ресурсы сразу после ее завершения. Кроме того, другая нить уже не может ожидать завершения отсоединенной нити. Другого влияния на выполнение нити отсоединение не оказывает. Нить можно также сразу создать отсоединенной, передав функции *pthread_create()* соответствующий сформированный набор атрибутов. Если же

создана обычная нить, ее можно отсоединить, вызвав из другой нити функцию *int pthread_detach(pthread_t thread)* и передав ей дескриптор отсоединяемой нити. Функция возвращает нуль, если отсоединение нити прошло успешно, и в противном случае возвращает код ошибки: EINVAL – переданный дескриптор не является дескриптором нити, которую можно присоединить; ESRCH – нет нити с таким дескриптором. Нить, которая была отсоединена, нельзя присоединить.

Досрочное завершение потока. Функции потоков можно рассматривать как вспомогательные программы, находящиеся под управлением функции *main* . Точно так же, как при управлении процессами, иногда возникает необходимость досрочно завершить процесс, многопоточной программе может понадобиться досрочно завершить один из потоков. Для досрочного завершения потока можно воспользоваться функцией *int pthread_cancel(pthread_t thread)*.

Единственным аргументом этой функции является идентификатор потока – *thread*. Функция *pthread_cancel()* возвращает 0 в случае успеха и ненулевое значение (код ошибки) в случае ошибки. Несмотря на то что *pthread_cancel()* может завершить поток досрочно, ее нельзя назвать средством принудительного завершения потоков. В теле функции потока можно не только самостоятельно выбрать порядок завершения в ответ на вызов *pthread_cancel()*, но и игнорировать этот вызов. Поэтому вызов функции *pthread_cancel()* следует рассматривать как запрос на выполнение досрочного завершения потока.

Функция *pthread_setcancelstate()* определяет, будет ли поток реагировать на обращение к нему с помощью *pthread_cancel()* или не будет. Сигнатура функции имеет вид *int pthread_setcancelstate(int state, int* oldstate)*.

Аргумент *state* может принимать два значения:

1) *PTHREAD_CANCEL_DISABLE* – запрет на завершения потока;

2) *PTHREAD_CANCEL_ENABLE* – разрешение на завершение потока.

Во второй аргумент *oldstate* записывается указатель на предыдущее значение аргумента *state*.

С помощью функции *pthread_setcancelstate()* можно указывать участки кода потока, во время исполнения которых поток нельзя завершить вызовом функции *pthread_cancel()*. Функция *pthread_testcancel()* создает точку возможного досрочного завершения потока (точку отмены потока). Такие точки необходимы для корректного завершения потока, так как, даже если досрочное завершение разрешено, поток, получивший запрос на досрочное завершение, часто может завершить работу не сразу. Если поток находится в режиме отложенного досрочного завершения (именно этот режим установлен по умолчанию), он выполнит запрос на досрочное завершение, только достигнув одной из точек отмены.

Сигнатура функции

pthread_testcancel(): void pthread_testcancel().

Тем не менее выполнение потока может быть прервано принудительно, не дожидаясь точек отмены. Для этого необходимо перевести поток в режим немедленного завершения, что делается с помощью вызова функции

pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL).

Вызов функции *pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL)* снова переводит поток в режим отложенного досрочного завершения.

Рассмотрим пример, в котором основной поток отправляет запрос на отмену потока, который он создал (рис. 1.1).

Когда вы выполните эту программу, то увидите следующий вывод (рис 1.2), демонстрирующий отмену потока.

После того как новый поток был создан обычным способом, основной поток засыпает (чтобы дать новому потоку время для запуска) и затем отправляет запрос на отмену потока (рис 1.3).

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
void *thread_function(void *arg);
int main() {
 int res;
 pthread_t a_thread;
 void *thread_result;
 res = pthread_create(&a_thread,
NULL, thread_function, NULL);
 if (res != 0) {
  perror(«Thread creation
failed»);
  exit(EXIT_FAILURE);
 }
 sleep(3);
 printf(«Canceling thread...\n»);
 res = pthread_cancel(a_thread);
 if (res != 0) {
  perror(«Thread cancelation
failed»);
  exit(EXIT_FAILURE);
 }
 printf(«Waiting for thread to
finish...\n»);
 res = pthread_join(a_thread,
&thread_result);
 if (res != 0) {
 perror(«Thread creation failed»);
  exit(EXIT_FAILURE);
 }
 sleep(3);
 printf(«Canceling thread...\n»);
 res = pthread_cancel(a_thread);
 if (res != 0) {
  perror(«Thread cancelation
failed»);
  exit(EXIT_FAILURE);
 }
 printf(«Waiting for thread to
finish...\n»);
 res = pthread_join(a_thread,
&thread_result);
 if (res != 0) {
  perror(«Thread join failed»);
  exit(EXIT_FAILURE);
 }
 exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
 int i, res;
 res = pthread_
setcancelstate(PTHREAD_
CANCEL_ENABLE, NULL);
 if (res != 0) {
  perror(“Thread pthread_
setcancelstate failed”);
  exit(EXIT_FAILURE);
 }
 res = pthread_
setcanceltype(PTHREAD_
CANCEL_DEFERRED, NULL);
 if (res != 0) {
  perror{“Thread pthread_
setcanceltype failed”);
  exit(EXIT_FAILURE);
 }
 printf(“thread_function is
running\n”);
 for(i = 0; i < 10; i++) {
  printf(“Thread is still running
(%d)...\n”, i);
  sleep(1);
 }
 pthread_exit(0);
}
```

Рис. 1.1

```
thread_function is running
Thread is still running (0)...
Thread is still running (1)...
Thread is still running (2)...
Canceling thread...
Waiting for thread to finish...
$
```

Рис. 1.2

```
sleep(3);
printf(«Cancelling thread...\n»);
res = pthread_cancel(a_thread);
if (res != 0) {
 perror(«Thread cancelation failed»);
 exit(EXIT_FAILURE);
}
```

Рис. 1.3

В созданном потоке вы сначала задаете состояние отмены, чтобы разрешить отмену потока, а далее (рис. 1.4) вы задаете тип отмены PTHREAD_CANCEL_DEFERRED, и в конце поток (рис. 1.5) ждет отмену.

```
res = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
if (res != 0) {
 perror("Thread pthread_setcancelstate failed");
 exit(EXIT_FAILURE);
}
res = pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED,
NULL);
if (res != 0) {
 perror("Thread pthread_setcanceltype failed");
 exit(EXIT_FAILURE);
}
```

Рис. 1.4

```
for (i = 0; i < 10; i++) {
 printf(«Thread is still running (%d)...\n», i);
 sleep(1);
}
```

Рис. 1.5

4. Компиляция многопоточных приложений

Для компиляции и сборки многопоточной программы необходимо иметь:

– стандартный компилятор *C* (*cc, gcc, g++*и т. д.);

– файлы заголовков: *<thread.h>*, *<pthread.h>*, *<errno.h>*, *<limits.h>*, *<signal.h>*, *<unistd.h>*;

– библиотеку реализации потоков (*libpthread*);

– другие библиотеки, совместимые с многопоточными приложениями (*libc*, *libm*, *libw*, *libintl*, *libnsl*, *libsocket*, *libmalloc*, *libmapmalloc*).

Файл заголовка *<pthread.h>*, используемый с библиотекой *lpthread*, компилирует код, который является совместимым с интерфейсами многопоточности, определенными стандартом *POSIX* 1003.1*c*.

Для компиляции программы, использующей потоки и реентерабельные системные функции, необходимо дополнительно указать в строке вызова компилятора следующие аргументы:

-D_REENTERANT –lpthread.

Команда компиляции *D* включает макрос *_REENTERANT*. Этот макрос указывает, что вместо обычных функций стандартной библиотеки к программе должны быть подключены их реентерабельные аналоги. Реентерабельный вариант библиотеки *glibc* написан таким образом, чтобы реализованные в ней реентерабельные функции как можно меньше отличались от их обычных аналогов. Также в строке вызова компилятора могут дополнительно указываться пути для поиска заголовочных файлов (ключ «*I*») и путь для поиска библиотек (ключ «*l*»). Для компоновщика указывается «*l*», это означает, что программа должна быть связана с библио-

текой *libpthread*, которая содержит все специальные функции, необходимые для работы с потоками.

5. Последовательность выполнения работы

1. Ознакомиться с теоретическим материалом.

2. Разработать алгоритм (в соответствии с полученным вариантом) и реализовать его в виде последовательной программы на *C* или *C++*.

3. Разработать тесты для программы и провести тестирование.

4. Выделить параллельные части, разработать параллельную многопоточную программу с использованием минимум двух дополнительных потоков по полученному варианту, использовать средства управления потоками.

5. Провести отладку на имеющихся тестах и доработать тестовый набор с учетом введенного параллелизма.

6. Убедиться в результативности применения многопоточности и средств управления потоками, сравнив результаты работы программ с их использованием.

Лабораторная работа № 17
СИНХРОНИЗАЦИЯ ДОСТУПА ПОТОКОВ К РАЗДЕЛЯЕМЫМ РЕСУРСАМ ПРОЦЕССА

1. Цель работы

Ознакомиться с подсистемой управления потоками в операционной системе *Linux* и основными программными механизмами для синхронизации доступа потоков к разделяемым ресурсам процесса.

2. Задание

Изучить основные программные средства синхронизации потоков в ОС *Linux*. Разработать приложения для многопоточных вычислений в соответствии с полученным вариантом.

3. Основные понятия

3.1. Механизмы синхронизации потоков

При выполнении нескольких потоков во многих случаях необходимо синхронизировать их взаимодействие с разделяемыми

ресурсами процесса. Существует несколько механизмов синхронизации потоков:

– взаимные исключения – мьютексы (MutEx – MUTial EXclusion);

– условные переменные состояния (Conditional Variable);

– семафоры.

Механизм использования условных переменных состояния и семафоров в многопоточных приложениях аналогичен механизму использования этих методов синхронизации в многопроцессных приложениях.

Механизм мьютексов представляет общий метод синхронизации выполнения потоков. Мьютекс можно определить как объект синхронизации, который устанавливается в особое сигнальное состояние, когда не занят каким-либо потоком. В любой момент мьютексом может владеть только один поток.

Использование мьютексов гарантирует, что только один поток в некоторый момент времени выполняет критическую секцию кода. Мьютексы можно использовать и в однопоточном коде. Доступны следующие действия с мьютексом: инициализация, удаление, захват или открытие, попытка захвата.

3.2. Программирование синхронизованного доступа потоков к разделяемым ресурсам с помощью мьютексов и условных переменных

Pthreads определяет набор типов и функций на языке программирования Си, где заголовочный файл – *pthread.h*.

Типы данных:

**pthread_mutex_t*: мьютекс;

**pthread_mutexattr_t*: атрибуты мьютекса;

**pthread_cond_t*: условная переменная;

**pthread_condattr_t*: атрибуты условной переменной.

Инициализация мьютекса. Функция *pthread_mutex_init()* предназначена для инициализации мьютекса int pthread_mutex_ init(pthread_mutex_t *mp, const pthread_mutexattr_t *mattr); мьютекс, на который указывает первый аргумент *mp*, инициализируется

значением по умолчанию, если второй аргумент *mattr* равен *NULL*, или определенными атрибутами, которые уже установлены с помощью *pthread_mutexattr_init()*.

Функция *pthread_mutex_init()* возвращает 0 после успешного завершения, или другое значение, если произошла ошибка. На рис. 2.1 – пример использования функции *pthread_mutexattr_init()* для инициализации мьютекса с произвольными значениями аттрибутов.

```
    #include <pthread.h>
    pthread_mutex_t mp
= PTHREAD_MUTEX_
INITIALIZER;
    pthread_mutexattr_t mattr;
    int ret;
    /* инициализация мьютекса
значением по умолчанию */
    ret = pthread_mutex_init(&mp,
NULL);
```

```
    ret = pthread_mutexattr_
init(&mattr);
    /* смена значений mattr с по-
мощью функций */
    ret = pthread_mutexattr_
setpshared(&mattr, PTHREAD_
PROCESS_SHARED);
    /* инициализация мьютекса
произвольными значениями */
    ret = pthread_mutex_init(&mp,
&mattr);
```

Рис. 2.1

Когда мьютекс инициализируется, он находится в открытом (разблокированном) состоянии. Статически определенные мьютексы могут инициализироваться непосредственно значениями по умолчанию с помощью константы *PTHREAD_MUTEX_INITIALIZER*.

Запирание (захват) мьютекса. Функция *pthread_mutex_lock()* используется для запирания или захвата мьютекса. Аргументом функции является адрес запираемого мьютекса. Если мьютекс уже заперт, вызывающий поток блокируется и мьютекс ставится в очередь приоритетов. Когда происходит возврат из *pthread_mute_lock()*, мьютекс запирается, а вызывающий поток становится его владельцем. Функция *pthread_mute_lock()* возвращает 0 после

успешного завершения, или другое значение, если произошла ошибка. Пример вызова:

```
#include <pthread.h>
pthread_mutex_t mp;
int ret;
ret = pthread_mutex_lock(&mp).
```

Для открытия (разблокировки) мьютекса используется функция *pthread_mutex_unlock()*. При этом мьютекс должен быть закрыт, а вызывающий поток должен быть владельцем мьютекса, т. е. тем, кто его запирал. Пока любые другие потоки ждут доступа к мьютексу, его поток-владелец, находящийся в начале очереди, не блокирован.

Функция *pthread_mutex_unlock()* возвращает 0 после успешного завершения, или другое значение, если произошла ошибка. Пример вызова:

```
#include <pthread.h>
pthread_mutex_t mp;
int ret;
ret = pthread_mutex_unlock(&mp);
```

Существует способ захвата мьютекса без блокирования потока (условной блокировки). Функция *pthread_mutex_trylock()* пытается провести запирание мьютекса. Она является неблокирующей версией *pthread_mutex_lock()*. Если мьютекс уже закрыт, вызов возвращает ошибку, однако поток, вызвавший эту функцию, не блокируется. В противном случае мьютекс закрывается, а вызывающий процесс становится его владельцем. Функция *pthread_ mutex_trylock()* возвращает 0 после успешного завершения, или другое значение, если произошла ошибка. Пример вызова:

```
#include <pthread.h>
pthread_mutex_t mp;
int ret;
ret = pthread_ mutex_trylock(&mp).
```

Захват через мьютекс не должен повторно инициализироваться или удаляться, пока другие потоки могут его использовать. Если

мьютекс инициализируется повторно или удаляется, приложение должно убедиться, что в настоящее время этот мьютекс не используется.

Удаление мьютекса. Функция *pthread_mutex_destroy()* используется для удаления мьютекса в любом состоянии. Функция *pthread_mutex_destroy()* возвращает 0 после успешного завершения, или другое значение, если произошла ошибка. Пример вызова:

```
#include <pthread.h>
pthread_mutex_t mp;
int ret;
ret = pthread_mutex_destroy(&mp).
```

Иерархия блокировок. Иногда может возникнуть необходимость одновременного доступа к нескольким ресурсам. При этом возникает проблема, рассмотренная в *примере А* и заключающаяся в том, что два потока пытаются захватить оба ресурса, но запирают соответствующие мьютексы в различном порядке.

Поток 1	Поток 2
/* намерен использовать ресурс_1 */ *pthread_mutex_lock(&m1)*; /* захватывает дополнительно ресурс_2+ресурс_1*/ *pthread_mutex_lock(&m2)*;	/* намерен использовать ресурс_2 */ *pthread_mutex_lock(&m2)* ; /* пытается захватить дополнительно ресурс_1+ресурс_2*/ *pthread_mutex_lock(&m1)*;

Рис. 2.2

В приведенном на рис. 2.2 *примере А* два потока запирают мьютексы *1* и *2*, связанные с соответствующими ресурсами, и возникает тупик при попытке запереть один из мьютексов. Наилучшим способом избежать проблем является запирание нескольких мьютексов в одном и том же порядке во всех потоках. Эта техника называется иерархией блокировок: мьютексы упорядочиваются

путем назначения каждому своего номера. После этого придерживаются правила – если мьютекс с номером *n* уже заперт, то нельзя запирать мьютекс с номером меньше *n*. Если блокировка всегда выполняется в указанном порядке, тупик не возникнет. Однако эта техника может использоваться не всегда, поскольку иногда требуется запирать мьютексы в порядке, отличном от порядка их номеров. Чтобы предотвратить тупик в этой ситуации, лучше использовать функцию *pthread_mutex_trylock()*. Один из потоков должен освободить свой мьютекс, если он обнаруживает, что может возникнуть тупик. На рис. 2.3 приведено использование условной блокировки *для примера А*.

// Поток 1	// Поток 2

pthread_mutex_lock(&m1);
pthread_mutex_lock(&m2);
/* обработка */
/* завершение обработки */
pthread_mutex_unlock(&m2);
pthread_mutex_unlock(&m1);
	for (; ;) {
	pthread_mutex_lock(&m2);
	if (pthread_mutex_
	trylock(&m1)==0)
	/* захват! */
	break;
	/* мьютекс уже заперт */
	pthread_mutex_unlock(&m2);
	}
	/* нет обработки */
	pthread_mutex_unlock(&m1);
	pthread_mutex_unlock(&m2);

Рис. 2.3

В том же *примере А* (рис. 2.3) поток *1* запирает мьютексы в нужном порядке, а поток *2* пытается закрыть их по-своему. Чтобы убедиться, что тупик не возникнет, поток *2* должен аккуратно обращаться с мьютексом *1*; если поток блокировался, ожидая мьютекс, который будет освобожден, он, вероятно, только что вызвал тупик с потоком *1*. Чтобы гарантировать, что этого не случится, поток *2* вызывает *pthread_mutex_trylock()*, который запирает мьютекс, если тот свободен. Если мьютекс уже заперт, поток *2*

получает сообщение об ошибке. В этом случае поток *2* должен освободить мьютекс *2*, чтобы поток *1* мог запереть его, а затем освободить оба мьютекса.

Рассмотрим *пример Б* (рис 2.4) использования мьютекса для контроля доступа к переменной. В приведенном ниже коде функция *increment_count()* использует мьютекс, чтобы гарантировать атомарность (целостность) модификации разделяемой переменной *count*. Функция *get_count()* использует мьютекс, чтобы гарантировать, что переменная *count* атомарно считывается.

```
#include <pthread.h>
pthread_mutex_t count_mutex;
long count;
void increment_count() {
pthread_mutex_lock(&count_
mutex);
count = count + 1;
pthread_mutex_unlock(&count_
mutex);
}
```

```
long get_count() {
long c;
pthread_mutex_lock(&count_
mutex);
c = count;
pthread_mutex_unlock(&count_
mutex);
return (c);
}
```

Рис. 2.4

Синхронизация с использованием условной переменной. Условная переменная позволяет потокам ожидать выполнения некоторого условия (события), связанного с разделяемыми данными. Над условными переменными определены две основные операции: информирование о наступлении события и ожидание события «на условной переменной». При выполнении операции «информирование» один из потоков, ожидающих значения условной переменной, возобновляет свою работу.

Условная переменная всегда используется совместно с мьютексом. Перед выполнением операции «ожидание» поток должен заблокировать мьютекс. При выполнении операции «ожидание» указанный мьютекс автоматически разблокируется. Перед возобновлением ожидающего потока выполняется автоматическая

блокировка мьютекса, позволяющая потоку войти в критическую секцию, после критической секции рекомендуется разблокировать мьютекс. При подаче сигнала другим потокам рекомендуется функцию «сигнализация» так же защитить мьютексом.

Прототипы функций для работы с условными переменными содержатся в файле *pthread.h* . Ниже приводятся прототипы функций вместе с пояснением их синтаксиса и выполняемых ими действий:

pthread_cond_init(pthread_cond_t cond, const pthread_ condattr_t* attr)* – инициализирует условную переменную *cond* с указанными атрибутами *attr* или с атрибутами по умолчанию (при указании 0 в качестве *attr*);

int pthread_cond_destroy(pthread_cond_t cond)* – уничтожает условную переменную *cond* ;

int pthread_cond_signal(pthread_cond_t cond)* – информирует о наступлении события те потоки, которые ожидают на условной переменной *cond* ;

*intpthread_cond_wait(pthread_cond_t*cond,pthread_mutex_t*mutex)*– переводит поток в состояние ожидания события «на условной переменной» *cond* .

Пример В. Ниже приведен фрагмент многопоточной программы синхронизации с использованием условных переменных (рис. 2.5), использующей мьютекс и условную переменную для синхронизации потока-производителя (*writer*) и потока-потребителя (*reader*), взаимодействующих через буфер *data* емкостью одна запись.

4. Компиляция многопоточных приложений

Для компиляции и сборки многопоточной программы необходимо иметь:

– стандартный компилятор *C* (*cc* , *gcc* , *g++* т. д.);

– файлы заголовков: *<thread.h>* , *<pthread.h>* , *<errno.h>* , *<limits.h>* , *<signal.h>* , *<unistd.h>* ;

– библиотеку реализации потоков (*libpthread*);

– другие библиотеки, совместимые с многопоточными приложениями (*libc*, *libm*, *libw*, *libintl*, *libnsl*, *libsocket*, *libmalloc*, *libmapmalloc*).

```
#include «main.h»
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <error.h>
…
int full;
pthread_mutex_t mx;
pthread_cond_t cond;
int data;
void *writer(void *)
{
while(1)
{
int t= write_to_buffer ();
pthread_mutex_lock(&mx)
while(full) {
pthread_cond_wait(&cond, &mx);
}
data=t;
full=1;
pthread_cond_signal(&mx);
pthread_mutex_unlock(&mx);
}
return NULL;
}
```

```
void * reader(void *)
{
while (1)
{
int t;
pthread_mutex_lock(&mx);
while (!full)
{
pthread_cond_wait(&cond,
&mx);
}
t=data;
full=0;
pthread_cond_signal(&mx);
pthread_mutex_unlock(&mx);
read_from _buffer();
}
return NULL;
}
```

Рис. 2.5

5. Последовательность выполнения работы:

1. Ознакомиться с теоретическим материалом.

2. Разработать алгоритм (в соответствии с полученным вариантом) и реализовать его в виде последовательной программы на *C* или *C++*.

3. Разработать тесты для программы и провести тестирование.

4. Выделить параллельные части, разработать параллельную многопоточную программу с использованием минимум двух дополнительных потоков по полученному варианту, использовать средства управления потоками.

5. Провести отладку на имеющихся тестах и доработать тестовый набор с учетом введенного параллелизма и убедиться в результативности применения многопоточности и средств синхронизации потоков, сравнив результаты работы программ с их использованием.

Лабораторная работа № 18
СИНХРОНИЗАЦИЯ ДОСТУПА ПОТОКОВ К РАЗДЕЛЯЕМЫМ РЕСУРСАМ ПРОЦЕССА С ПОМОЩЬЮ МОНИТОРОВ

1. Цель работы

Ознакомиться с подсистемой управления потоками в операционной системе *Linux* и высокоуровневыми программными механизмами для синхронизации доступа потоков к разделяемым ресурсам процесса.

2. Задание

Изучить основные программные средства синхронизации потоков в ОС *Linux*. Разработать программный монитор и внешнее приложение в соответствии с полученным вариантом.

3. Основные понятия

3.1. Понятие программного монитора

Мониторы являются программными модулями и используются для того, чтобы сгруппировать представление и реализацию разделяемого ресурса (класса). Монитор состоит из интерфейса и тела. Интерфейс определяет предоставляемые ресурсом операции (методы). Тело содержит переменные, описывающие состояние ресурса, и процедуры, реализующие операции интерфейса. Таким образом, монитор является статическим объектом вида:

monitor name {
[*объявления постоянных переменных*]
[*операторы инициализации*]
[*процедуры*]
}.

Программист монитора может не знать заранее, в каком порядке будут поступать запросы на выполнение процедур от внешнего приложения. В связи с этим полезно определить предикат, сохраняющий истинное значение независимо от порядка выполнения запросов. *Инвариант монитора* – это предикат, определяющий «разумные» состояния постоянных переменных, когда процессы не обращаются к ним. Код инициализации монитора должен создать состояние переменной, соответствующее инварианту, а каждая процедура должна его поддерживать.

Внешний процесс для получения доступа к разделяемому ресурсу вызывает нужную процедуру монитора. Пока некоторый процесс выполняет операторы процедуры, она *активна.* В любой момент времени может быть активным только один экземпляр только одной процедуры монитора, т. е. одновременно не могут быть активными ни два вызова разных процедур, ни два вызова одной и той же процедуры.

3.2. Программирование монитора

Рассмотрим в качестве примера разработанный на *C++* монитор и программу внешнего процесса, обращающуюся с запросами к монитору. Моделируется управление банковским счетом с помощью инструмента синхронизации высокого уровня – монитора. Существует четыре вида внешних запросов для обработки разделяемого ресурса, которым является текущее значение банковского счета:

1) «проверка состояния»;
2) «перевод на счет»;
3) «снятие со счета»;
4) «отложенное снятие со счета».

Запрос отложенного снятия «ожидает» на условной переменной монитора и выполняется после появления требуемой суммы на счете. Для упрощения полагают, что в каждый момент времени может быть не более одного отложенного запроса на снятие со счета. Потоки, модифицирующие запросы на изменение состояния счета, выполняются в режиме взаимного исключения. Запрос-поток «проверка состояния» не синхронизируется с другими потоками и может быть запущен в любой момент. Монитор счета представляет собой класс с соответствующими методами (рис. 3.1; 3.2).

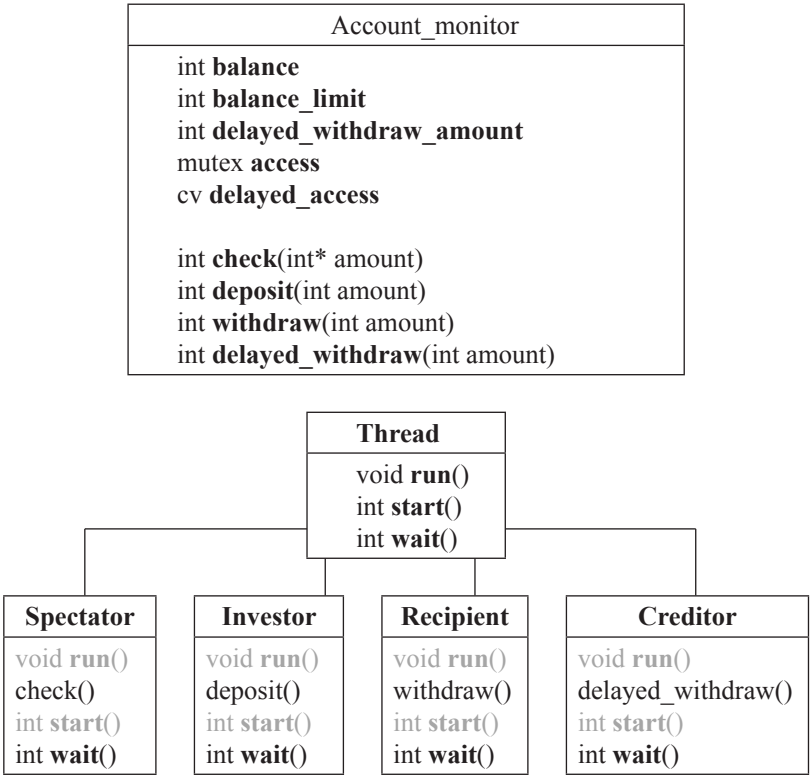

Рис. 3.1. Диаграмма классов проекта

Поля	Методы
• int **balance** – текущий баланс на счете; • int **balance_limit** – лимит счета (максимальное значение баланса); • int **delayed_withdraw_amount** – сумма, необходимая для выполнения отложенного вывода со счета; • mutex **access** – мьютекс для доступа к балансу; • cv **delayed_access** – условная переменная, регулирующая выполнение метода **delayed_withdraw()**.	**Account_monitor**(int balance_limit, int* retval) – конструктор. Получает в качестве аргумента значение лимита счета. Записывает код возврата по адресу **retval**; • int **check**(int* amount) – проверка состояния счета. Записывает текущее значение баланса по адресу **amount**; • int **deposit**(int amount) – перевод средств в размере **amount** на счет. Освобождает условную переменную **delayed_access**; • int **withdraw**(int amount) – вывод средств в размере **amount** со счета; • int **delayed_withdraw**(int amount) – отложенный вывод средств в размере **amount** со счета. Регулируется условной переменной **delayed_access**.

Рис. 3.2. Описание классов **Account_monitor**.
Монитор банковского счета

Инвариантом данного монитора является предикат, содержащий условия неотрицательности текущего баланса счета и непревышения лимита текущего баланса счета. Проверки инварианта выполняются во всех методах класса **Account_monitor**. В случае обнаружения несоответствия возвращается код *критической ошибки* (рис. 3.3).

Thread Абстрактный класс, описывающий поток.

Методы:

void **run**() – непосредственно функция потока.

int **start**() – запуск потока. Содержит вызов **pthread_create()**. Возвращает 0 в случае успеха.

int **wait**() – ожидание завершения данного потока вызывающим потоком. Возвращает 0 в случае успеха.

Spectator. Наблюдатель. Проверяет состояние счета. Наследуется от класса **Thread**.

run() использует метод **check()** класса **Account_monitor**.

Investor. Вкладчик. Выполняет перевод средств на счет. Наследуется от класса **Thread**.

run() использует метод **deposit()** класса **Account_monitor**.

Recipient. Получатель. Снимает средства со счета. Наследуется от класса **Thread**.

run() использует метод **withdraw()** класса **Account_monitor**.

Creditor. Кредитор. Списывает средства со счета при их поступлении. Наследуется от класса **Thread**.

run() использует метод **delayed_withdraw()** класса **Account_monitor**.

SUCCESS	Успешное выполнение
WRONG_LIMIT_VALUE	Неправильное (неположительное) значение лимита счета при инициализации монитора
BAD_CHECK_POINTER	Неправильный указатель в аргументах метода **check**()
WRONG_AMOUNT_VALUE	Неправильное (неположительное) значение средств для текущей операции
TOO_MUCH_AMOUNT	Слишком большое значение средств для текущей операциии (перевод или вывод)
DELAYED_WITHDRAW_ACTIVE	Отложенный платеж уже активирован
BALANCE_NEGATIVE	**Критическая ошибка**. Отрицательный баланс на счете
LIMIT_EXCEEDED	**Критическая ошибка**. Превышен лимит счета

Рис. 3.3. Коды возврата методов

Конструкторы классов, реализующих потоки, получают при вызове указателя на объект типа **Account_monitor**. Конструкторы классов потоков, модифицирующих состояние счета, также получают значения сумм средств для добавления или снятия. Представлены на рис. 3.4 и 3.5.

account.h

```
#include <pthread.h>
#include <cstdlib>

using namespace std;

#ifndef _ACCOUNT_MONITOR
#define _ACCOUNT_MONITOR

// return values
#define SUCCESS 0
#define WRONG_LIMIT_VALUE 1
#define BAD_CHECK_POINTER 2
#define BALANCE_NEGATIVE 3
#define LIMIT_EXCEEDED 4
#define WRONG_AMOUNT_VALUE 5
#define TOO_MUCH_AMOUNT 6
#define DELAYED_WITHDRAW_ACTIVE 7

class Account_monitor {
private:
int balance_;
int balance_limit_;
int delayed_withdraw_amount_;
pthread_mutex_t access_;
pthread_cond_t delayed_access_cv_;
public:
Account_monitor(int balance_limit, int* retval);
int check(int* amount);
int deposit(int amount);
int withdraw(int amount);
int delayed_withdraw(int amount);
};
```

```
Account_monitor::Account_monitor(int balance_limit, int* retval) {
if (balance_limit <= 0) {
(*retval) = WRONG_LIMIT_VALUE;
}
else {
balance_ = 0;
balance_limit_ = balance_limit;
delayed_withdraw_amount_ = 0;
pthread_mutex_init(&access_, NULL);
pthread_cond_init(&delayed_access_cv_, NULL);
(*retval) = SUCCESS;
}
}
```

```
int
Account_monitor::check(int* amount) {
if(amount == NULL)
return BAD_CHECK_POINTER;
if(balance_ < 0)
return BALANCE_NEGATIVE;
if(balance_ > balance_limit_)
return LIMIT_EXCEEDED;
(*amount) = balance_;
return SUCCESS;
}
```

```
int
Account_monitor::deposit(int amount) {
if(amount <= 0)
return WRONG_AMOUNT_VALUE;
if(amount > balance_limit_)
return TOO_MUCH_AMOUNT;
pthread_mutex_lock(&access_);
sleep(2); // delay
int new_balance = balance_ + amount;
if( new_balance <= balance_limit_ ) {
balance_ = new_balance;
```

```
if(delayed_withdraw_amount_ <= balance_)
pthread_cond_signal(&delayed_access_cv_); // to delayed_withdraw
pthread_mutex_unlock(&access_);
return SUCCESS;
}
else {
pthread_mutex_unlock(&access_);
return TOO_MUCH_AMOUNT;
}
}
```

```
int
Account_monitor::withdraw(int amount) {
if(amount <= 0)
return WRONG_AMOUNT_VALUE;
if(amount > balance_limit_)
return TOO_MUCH_AMOUNT;
pthread_mutex_lock(&access_);
sleep(2); // delay
int new_balance = balance_ – amount;
if( new_balance >= 0) {
balance_ = new_balance;
pthread_mutex_unlock(&access_);
return SUCCESS;
}
else{
pthread_mutex_unlock(&access_);
return TOO_MUCH_AMOUNT;
}
}
```

```
int
Account_monitor::delayed_withdraw(int amount) {
if(amount <= 0)
return WRONG_AMOUNT_VALUE;
if(amount > balance_limit_)
return TOO_MUCH_AMOUNT;
```

```
pthread_mutex_lock(&access_);
if(delayed_withdraw_amount_ == 0) {
if(amount >= balance_) {
delayed_withdraw_amount_ = amount;
pthread_cond_wait(&delayed_access_cv_, &access_); // wait cv
}
sleep(2); // delay
balance_ -= amount;
delayed_withdraw_amount_ = 0;
pthread_mutex_unlock(&access_);
return SUCCESS;
}
else {
pthread_mutex_unlock(&access_);
return DELAYED_WITHDRAW_ACTIVE;
}
}

#endif
```

Рис. 3.4. Исходные тексты программ монитора Account.h

transactions.h

```
#include <iostream>
#include “account.h”

using namespace std;

#ifdef _ACCOUNT_MONITOR
#ifndef _TRANSACTIONS
#define _TRANSACTIONS

// transaction return code output
char* trcode(int code) {
switch(code) {
case 0: return “SUCCESS”;
case 1: return “WRONG_LIMIT_VALUE”;
case 2: return “BAD_CHECK_POINTER”;
```

```
    case 3: return “BALANCE_NEGATIVE”;
    case 4: return “LIMIT_EXCEEDED”;
    case 5: return “WRONG_AMOUNT_VALUE”;
    case 6: return “TOO_MUCH_AMOUNT”;
    case 7: return “DELAYED_WITHDRAW_ACTIVE”;
    default: return “UNKNOWN_STATE”;
    }
    }

    // basic thread for transaction
    class Thread
    {
    private:
    pthread_t thread;
    Thread(const Thread& copy); // copy constructor denied
    static void *thread_func(void *d) { ((Thread *)d)->run(); }
    public:
    Thread() {}
    virtual ~Thread() {}
    virtual void run() = 0;
    int start() { return pthread_create(&thread, NULL,
    Thread::thread_func, (void*)this); }
    int wait () { return pthread_join (thread, NULL); }
    };
```

```
    // investor ( uses deposit() )
    class Investor:public Thread {
    private:
    Account_monitor* account_;
    int amount_;
    public:
    Investor(Account_monitor* account, int amount) {
    account_ = account;
    amount_ = amount;
    }
    void run() {
    int retval = account_->deposit(amount_);
    cout << “Deposit : status=” << trcode(retval) << “ amount=$” <<
amount_ << ‘\n’;
    }
    };
```

```
    // recipient ( uses withdraw() )
    class Recipient:public Thread {
    private:
    Account_monitor* account_;
    int amount_;
    public:
    Recipient(Account_monitor* account, int amount) {
    account_ = account;
    amount_ = amount;
    }
    void run() {
    int retval = account_->withdraw(amount_);
    cout << “Withdraw : status=” << trcode(retval) << “ amount=$” <<
amount_ << ‘\n’;
    }
    };
```

```
    // creditor ( uses delayed_withdraw() )
    class Creditor:public Thread {
    private:
    Account_monitor* account_;
    int amount_;
    public:
    Creditor(Account_monitor* account, int amount) {
    account_ = account;
    amount_ = amount;
    }
    void run() {
    int retval = account_->delayed_withdraw(amount_);
    cout << “Withdraw (d) : status=” << trcode(retval) << “ amount=$” <<
amount_ << ‘\n’;
    }
    };
```

```
    // spectator ( uses check() )
    class Spectator:public Thread {
    private:
    Account_monitor* account_;
    public:
    Spectator(Account_monitor* account) {
```

```
    account_ = account;
    }
    void run() {
    int amount;
    int retval = account_->check(&amount);
    cout << "Check : status=" << trcode(retval) << " amount=$" << amount
<< '\n';
    }
    };

    #endif
    #endif
```

bank-account.cpp

```
    #include "transactions.h"

    #ifdef _ACCOUNT_MONITOR
    #ifdef _TRANSACTIONS

    const int balance_limit = 1000;
    int retval = 0;
    Account_monitor account(balance_limit, &retval);

    int main(void) {
    if(retval != 0) {
    trcode(retval);
    cout << "\nEXIT_FAILURE\n";
    return EXIT_FAILURE;
    }
    int amount;
    Investor* investor;
    Recipient* recipient;
    Creditor* creditor;
    Spectator* spectator;
    while(true) {
    cin >> amount;
    if(amount > 0) {
    investor = new Investor(&account, amount);
```

```
investor->start();
}
if(amount < 0) {
if(amount > -100) {
recipient = new Recipient(&account, -amount);
recipient->start();
}
else {
creditor = new Creditor(&account, -amount);
creditor->start();
}
}
if(amount == 0) {
spectator = new Spectator(&account);
spectator->start();
}
}
investor->wait();
recipient->wait();
spectator->wait();
return EXIT_SUCCESS;
}

#endif
#endif
```

Рис. 3.5. Исходные тексты программ

4. Компиляция многопоточных приложений

Для компиляции и сборки многопоточной программы необходимо иметь:

– стандартный компилятор *C++* (*g++* т. д.);

– файлы заголовков: <*thread.h*> , <*pthread.h*> , <*errno.h*> , <*limits.h*> , <*signal.h*> , <*unistd.h*> ;

– библиотеку реализации потоков (*libpthread*).

5. Последовательность выполнения работы:

1. Ознакомиться с теоретическим материалом.

2. Разработать алгоритм и определить инвариант (в соответствии с полученным вариантом), реализовать его в виде программы на *C++*.

3. Разработать тесты для программы и провести тестирование.

4. Разработать параллельную многопоточную программу-монитор и внешнюю программу с использованием дополнительных потоков по полученному варианту, использовать средства синхронизации потоков.

5. Провести отладку на имеющихся тестах и доработать тестовый набор с учетом введенного параллелизма и убедиться в результативности применения многопоточности и средств синхронизации потоков.

ТРЕБОВАНИЯ К ОТЧЕТАМ

По результатам выполнения лабораторных работ необходимо составить отчет. Рекомендуется составлять отчет параллельно с выполнением заданий лабораторных работ, не оставляя этот процесс «на потом». Отчет должен начинаться с титульного листа установленного образца, с названием университета, института высшей школы или кафедры. На титульном листе указывается ф. и. о. студента-исполнителя работы, номер студенческой группы и ф. и. о. преподавателя.

В отчете приводится цель работы и формулировки заданий (фрагменты текста заданий копируются из электронной версии данного учебного пособия), вслед за которыми помещается информация, подтверждающая выполнение студентом заданий лабораторных работ: скриншоты с результатами исполнения команд и программ, исходные тексты тех *C*-программ (с комментариями) или фрагментов, которые были составлены самим студентом, скрипты (*BASH* и др.) командных файлов, дампы памяти и пакетов. Отчет должен содержать ответы на поставленные в заданиях вопросы и выводы по работе.

В выводах по работам части 2 должны приводиться результаты работы программ при тестировании с использованием средств

синхронизации потоков и без их использования, а также пояснения полученных различий между результатами работы программ при использовании многопоточности и средств управления потоками.

Защита лабораторных работ происходит по предъявлению оформленного отчета и сопровождается демонстрацией исполнения программ и командных файлов, а также ответами на вопросы преподавателя.

Отчет можно создавать под ОС *Linux*, например, с помощью приложения *LibreOffice Writer* , сохраняя в Home-каталоге вашего рабочего компьютера.

Библиографический список

1. **Барабанова М. И.** Информационные технологии: открытые системы, сети, безопасность в системах и сетях : учеб. пособие / М. И. Барабанова, В. И. Кияев. – СПб. : Изд-во СПбГУЭФ, 2010.

2. **Робачевский А.** Операционная система UNIX / А. Робачевский. – СПб. : БХВ-Петербург, 1999.

3. **Стивенс У.** UNIX : взаимодействие процессов / У. Стивенс. – СПб. : Питер, 2002.

4. **Тейнсли Д.** Linux и UNIX : программирование в shell. Руководство разработчика / Д. Тейнсли. – Киев : Издательская группа BHV, 2001.

5. **Чан Т.** Системное программирование на C++ для UNIX / Т. Чан. – Киев : Издательская группа BHV, 1999.

6. **Эндрюс Г. Р.** Основы многопоточного, параллельного и распределенного программирования / Г. Р. Эндрюс. – М. : ИД «Вильямс», 2003.

7. **Brown C.** UNIX Distributed Programming. Prentice Hall International (UK) Limited, 1994.

Шмаков Владимир Эдуардович
Хлудова Марина Васильевна

ОТКРЫТЫЕ СИСТЕМЫ И LINUX-ТЕХНОЛОГИИ

Учебное пособие

Редактор *Е. В. Емяшева*
Корректор *Н. Б. Цветкова*
Компьютерная верстка *Е. Г. Орловского*

Санитарно-эпидемиологическое заключение
№ 78.01.07.953.П001342.01.07 от 24.01.2007 г.

Налоговая льгота – Общероссийский классификатор продукции
ОК 005-93, т. 2; 95 3005 – учебная литература

Подписано в печать 25.04.2018. Формат 60×84/16.
Усл. печ. л. 3,6. Тираж 50 экз. Заказ 58.

Отпечатано с оригинал-макета
в Издательско-полиграфическом центре Политехнического университета.
195251, Санкт-Петербург, Политехническая ул., 29.
Тел.: (812) 552-77-17; 550-40-14.