

**Министерство образования и науки России
Государственное образовательное учреждение
высшего образования
«Казанский национальный исследовательский
технологический университет»**

Н. А. Староверова, Э. П. Ибрагимова

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Учебное пособие

**Казань
Издательство КНИТУ
2016**

УДК 004.451

ББК

Н.А. Староверова

Операционные системы: учебное пособие / Н.А. Староверова, Э.П. Ибрагимова. – Минобрнауки России, Казань: Изд-во **КНИТУ**, 2016.
ISBN

Пособие соответствует требованиям Федерального государственного образовательного стандарта высшего образования (№ 41030 от 09.02.2016) направления бакалаврской подготовки 09.03.01 «Информатика и вычислительная техника».

Содержит основную информацию по дисциплинам «Операционные системы» и «Системное программное обеспечение». Рассмотрены принципы организации системного программирования и особенности программирования процессов в UNIX-подобных операционных системах. Основной акцент делается на изучение организации управления процессами на примере ОС Linux, которая является многопроцессной UNIX-подобной операционной системой.

Предназначено для студентов очного и заочного отделения факультета автоматизации и управления, изучающих дисциплины «Операционные системы» и «Системное программное обеспечение» в рамках бакалаврской подготовки.

Подготовлено на кафедре «Автоматизированные системы сбора и обработки информации».

Печатается по решению редакционно-издательского совета Казанского национального исследовательского технологического университета.

Рецензенты:

доц. кафедры АТПП ФБГОУ ВПО «КГЭУ» С. В. Карпеев

доц. кафедры АТПП ФБГОУ ВПО «КГЭУ» В. В. Плотников

ISBN

© Староверова Н.А., Ибрагимова Э.П. 2016.

© Казанский национальный исследовательский технологический университет, 2016.

ОГЛАВЛЕНИЕ

Введение	4
Лабораторная работа 1. Знакомство с операционной системой UNIX.....	5
Лабораторная работа 2. Администрирование.....	26
Лабораторная работа 3. Процессы в операционной системе UNIX	44
Лабораторная работа 4. Организация взаимодействия процессов с помощью каналов	68
Лабораторная работа 5. Средства SYSTEM IPC. Организация работы с разделяемой памятью в UNIX. Понятие нитей исполнения (thread).....	101
Лабораторная работа 6. Семафоры в UNIX как средство синхронизации процессов.	140
Лабораторная работа 7. Очереди сообщений.....	164
Лабораторная работа 8. Организация файловой системы в UNIX. Работа с файлами и директориями. Понятие MEMORY MAPPED.	185
Лабораторная работа 9. Организация ввода-вывода в UNIX. Файлы устройств. Аппарат прерываний. Сигналы в UNIX. ...	221
Лабораторная работа 10. Семейство протоколов TCP/IP. Сокеты (sockets) в UNIX и основы работы с ними.	257

ВВЕДЕНИЕ

Операционная система (ОС) - это комплекс взаимосвязанных программ, предназначенных для управления ресурсами вычислительного устройства и организации взаимодействия с пользователем.

На сегодняшний день существует довольно большое количество различных операционных систем. От долгожителей, таких как UNIX и его клоны, до совсем новых и малоизвестных систем.

Исследовав различные современные операционные системы, можно выделить следующие основные направления развития ОС.

- Графические оболочки.
- Поддержка новых сетевых технологий и Web-технологий.
- Усиленное внимание к механизмам безопасности и защиты.
- Поддержка многопоточности и многоядерных процессоров.
- Поддержка распределенных и параллельных вычислений.
- Виртуализация ресурсов и аппаратуры.
- Развитие файловых систем.
- Поддержка облачных вычислений

Дисциплина «Операционные системы» предназначена для изучения принципов организации и освоения основ системного программирования и особенностей программирования процессов в UNIX-подобных операционных системах. Она обеспечивает фундамент для изучения всех профильных дисциплин, преподаваемых в рамках направления «Информатика и вычислительная техника».

В учебном пособии рассматриваются 10 тем, каждая содержит примеры программ и практические задания. Основной акцент делается на изучение организации управления процессами на примере ОС Linux, которая является многопроцессной UNIX-подобной операционной системой. Данная ОС представляет собой систему с открытым кодом, что обеспечивает возможность использования ее в образовательных учреждениях.

Пособие построено таким образом, что сначала происходит знакомство с операционной системой, рассматриваются основные вопросы администрирования, а затем студенты знакомятся с созданием процессов и основными принципами межпроцессного взаимодействия. В каждой лабораторной работе приводятся примеры кода, составленного в C и дающего возможность на практике ознакомиться с

рассмотренными теоретическими вопросами. Кроме того, студентам предлагается большой выбор заданий, который они могут выполнить в домашних условиях.

Обширный теоретический и практический материал, представленный в пособии, поможет студентам очного и заочного отделения факультета управления и автоматизации в эффективном освоении курсов «Операционные системы» и «Системное программное обеспечение».

Лабораторная работа 1

ЗНАКОМСТВО С ОПЕРАЦИОННОЙ СИСТЕМОЙ UNIX

Цели и задачи

Начальное знакомство с системой, вход в систему, работа в терминальном режиме, изучение основных команд UNIX, получение начальных сведений о структуре каталогов в UNIX. Работа со справочной системой. Удаленный вход в систему. Вход в систему, работа в терминальном режиме, изучение основных команд UNIX.

1.1 Вход с системной консоли

Вход в систему осуществляется с системной консоли, которая представляет собой монитор и клавиатуру, связанные непосредственно с системой. Как многопользовательская система UNIX предоставляет возможность работы в нескольких виртуальных символьных терминалах (виртуальных консолях), которые позволяют запускать программы в разных терминалах и от имени разных пользователей работать одновременно под несколькими именами или под одним именем и т.п.

Максимально возможное количество виртуальных терминалов – 12, по умолчанию установленная система представляет 6 виртуальных символьных терминалов и один графический. Переключение между терминалами осуществляется комбинацией клавиш **<Alt> – <F1>** – первый терминал, **<Alt> – <F2>** – второй терминал и т.д. Переключение из графического терминала в символьный осуществляется сочетанием трех функциональных клавиш **<Ctrl> – <Alt> – <F#>**, где # – номер символьного терминала.

При входе в систему на конкретном терминале пользователь видит

приглашение **hostname login**, где **hostname** – имя машины, на которой регистрируется пользователь.

После успешного ввода имени пользователя и пароля система выводит приглашение к вводу команды.

– для суперпользователя **root**;

\$ – для всех остальных пользователей.

Система готова к вводу команды, и пользователь может запустить утилиту **mc**, которая является удобной оболочкой работы с файловой системой.

\$ mc

Часто при первом входе в систему пользователя требуется поменять пароль, назначенный пользователю администратором, – используйте команду **passwd**.

\$ passwd

Выход из терминала осуществляется по команде **exit**

\$ exit

1.2 Понятия **login** и **password**

Операционная система UNIX является многопользовательской операционной системой. Для обеспечения безопасной работы пользователей и целостности системы доступ к ней должен быть санкционирован. Для каждого пользователя, которому разрешен вход в систему, заводится специальное регистрационное имя – **username** или **login** – и сохраняется специальный пароль – **password**, соответствующий этому имени. Как правило, при регистрации нового пользователя начальное значение пароля для него задает системный администратор. После первого входа в систему пользователь должен изменить начальное значение пароля с помощью специальной команды. В дальнейшем он может в любой момент изменить пароль по своему желанию.

1.3 Упрощенное понятие об устройстве файловой системы в UNIX.

Полные и относительные имена файлов

Понятие «файл» характеризует статическую сторону вычислительной системы. Все файлы, доступные в операционной системе UNIX, как и в уже известных вам операционных системах,

объединяются в древовидную логическую структуру. Файлы могут объединяться в каталоги или директории. Не существует файлов, которые не входили бы в состав какой-либо директории. Директории, в свою очередь, могут входить в состав других директорий. Допускается существование пустых директорий, в которые не входит ни один файл, и ни одна другая директория (рис. 1.1). Среди всех директорий существует только одна директория, которая не входит в состав других директорий, – ее принято называть корневой. На настоящем уровне недостаточного знания UNIX можно заключить, что в файловой системе UNIX присутствует, по крайней мере, два типа файлов: обычные файлы, которые могут содержать тексты программ, исполняемый код, данные и т.д. (их принято называть регулярными файлами), и директории.

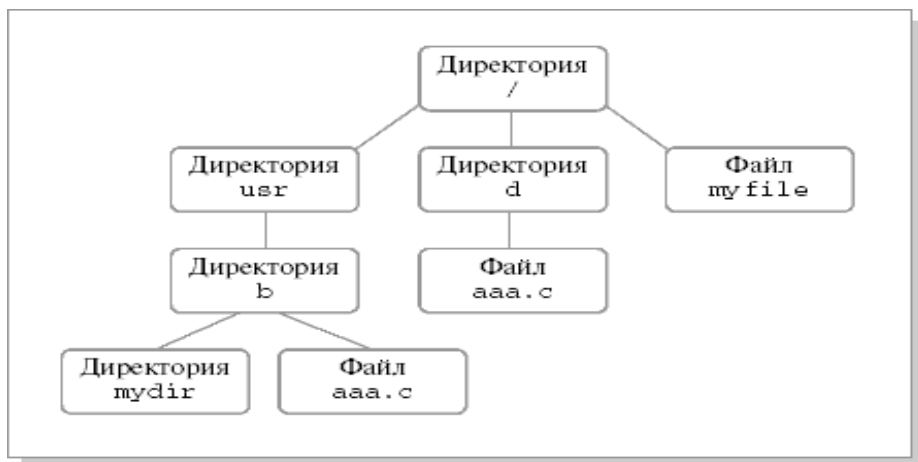


Рис. 1.1. Пример структуры файловой системы

Каждому файлу (регулярному или директории) должно быть присвоено имя. В различных версиях операционной системы UNIX существуют те или иные ограничения на построение имени файла. В стандарте POSIX на интерфейс системных вызовов для операционной системы UNIX содержится лишь три явных ограничения:

- Нельзя создавать имена большей длины, чем это предусмотрено операционной системой (для Linux – 255 символов).

- Нельзя использовать символ NUL (не путать с указателем NULL!) – он же символ с нулевым кодом, он же признак конца строки в языке Си.
- Нельзя использовать символ '/'.

Единственным исключением является корневая директория, которая всегда имеет имя "/". Эта же директория представляет собой единственный файл, который должен иметь уникальное имя во всей файловой системе. Для всех остальных файлов имена должны быть уникальными только в рамках той директории, в которую они непосредственно входят. Каким же образом отличить два файла с именами «aaa.c», входящими в директории «b» и «d» на рисунке 1.1, что было понятно о каком из них идет речь? Здесь на помощь приходит понятие полного имени файла. Мысленно построим путь от корневой вершины дерева файлов к интересующему нас файлу и выпишем все имена файлов (т.е. узлов дерева), встречающиеся на нашем пути, например, `"/usr/b/aaa.c"`. В этой последовательности первым будет всегда стоять имя корневой директории, а последним – имя интересующего нас файла. Отделим имена узлов друг от друга в этой записи не пробелами, а символами `"/"`, за исключением имени корневой директории и следующего за ним имени (`"/usr/b/aaa.c"`). Полученная запись однозначно идентифицирует файл во всей логической конструкции файловой системы. Такая запись и получила название полного имени файла.

1.4 Понятие текущей директории. Команда `pwd`. Относительные имена файлов

Для каждой работающей программы в операционной системе, включая командный интерпретатор (shell), который обрабатывает вводимые команды и высвечивает приглашение к их вводу, одна из директорий в логической структуре файловой системы назначается текущей или рабочей для данной программы. Узнать, какая директория является текущей для вашего командного интерпретатора, можно с помощью команды операционной системы `pwd`.

Домашняя директория пользователя и ее определение. Для каждого нового пользователя в системе заводится специальная директория, которая становится текущей сразу после его входа в систему. Эта директория получила название домашней директории

пользователя. Воспользуйтесь командой *pwd* для определения своей домашней директории.

1.5 Команда *man* – универсальный справочник

По ходу изучения операционной системы UNIX вам часто будет требоваться информация о том, что делает та или иная команда или системный вызов, какие у них параметры и опции, для чего предназначены некоторые системные файлы, каков их формат и т. д. Большая часть информации в UNIX Manual доступна в интерактивном режиме с помощью утилиты *man*.

Пользоваться утилитой *man* достаточно просто: наберите команду

\$ ***man имя***, где «*имя*» – это имя интересующей вас команды, утилиты, системного вызова, библиотечной функции или файла. Посмотрите с ее помощью информацию о команде *pwd*.

Чтобы пролистать страницу полученного описания, если оно не поместилось на экране полностью, следует нажать клавишу **<пробел>**. Для прокрутки одной строки воспользуйтесь клавишей **<Enter>**. Вернуться на страницу назад позволит одновременное нажатие клавиш **<Ctrl>** и ****. Выйти из режима просмотра информации можно с помощью клавиши **<q>**.

1.6 Команды *cd* для смены текущей директории и *ls* для просмотра состава директории

Для смены текущей директории командного интерпретатора можно воспользоваться командой *cd* (change directory). Для этого необходимо набрать команду в виде

\$ ***cd имя_директории***, где «*имя_директории*» – полное или относительное имя директории, которую вы хотите сделать текущей. Команда *cd* без параметров сделает текущей директорией вашу домашнюю директорию.

Просмотреть содержимое текущей или любой другой директории можно, воспользовавшись командой *ls* (от list). Если ввести ее без параметров, эта команда распечатает вам список файлов, находящихся в текущей директории. Если же в качестве параметра задать полное или относительное имя директории:

\$ *ls имя_директории*, – то она распечатает список файлов в указанной директории. Надо отметить, что в полученный список не войдут файлы, имена которых начинаются с символа «точка» – «.». Такие файлы обычно создаются различными системными программами для своих целей (например, для настройки).

Посмотреть полный список файлов можно, дополнительно указав команде ls опцию -a, т.е. набрав ее в виде

\$ *ls -a* или \$ *ls -a имя_директории*

У команды ls существует и много других опций.

Команда ls с опциями -al. Позволяет получить подробную информацию о файлах в некоторой директории, включая имена хозяина, группы хозяев и права доступа, можно с помощью уже известной нам команды ls с опциями -al.

\$ *ls -al*

В выдаче этой команды третья колонка слева содержит имена пользователей хозяев файлов, а четвертая колонка слева – имена групп хозяев файла. Крайняя левая колонка содержит типы файлов и права доступа к ним. Тип файла определяет первый символ в наборе символов. Если это символ 'd', то тип файла – директория, если там стоит символ '-', то это регулярный файл. Следующие три символа определяют права доступа для хозяина файла, следующие три – для пользователей, входящих в группу хозяев файла, и последние три – для всех остальных пользователей. Наличие символа (r, w или x), соответствующего праву, для некоторой категории пользователей означает, что данная категория пользователей обладает этим правом.

Для получения полной информации о команде ls воспользуйтесь утилитой man.

1.7 Команда cat и создание файла. Перенаправление ввода и вывода

Вы уже умеете перемещаться по логической структуре файловой системы и рассматривать ее содержимое. Следует уметь также и просматривать содержимое файлов, и создавать их. Для просмотра содержимого небольшого текстового файла на экране можно воспользоваться командой cat.

Если набрать ее в виде

\$ *cat имя_файла*, то на экран выведется все его содержимое.

Если ваш текстовый файл большой, то вы увидите только его последнюю страницу. Большой текстовый файл удобнее рассматривать с помощью утилиты more (описание ее использования вы найдете в UNIX Manual).

Если в качестве параметров для команды cat задать не одно имя, а имена нескольких файлов:

\$ *cat файл1 файл2 ... файлN*, то система выдаст на экран их содержимое в указанном порядке. Вывод команды cat можно перенаправить с экрана терминала в какой-нибудь файл, воспользовавшись символом перенаправления выходного потока данных – знаком «>» («больше»).

Команда \$ *cat файл1 файл2 ... файлN > файл_результата* сольет содержимое всех файлов, чьи имена стоят перед знаком ">", воедино в файл_результата – конкатенирует их (от англ. ‘concatenate’ – объединять – и произошло название команды).

Прием перенаправления выходных данных со стандартного потока вывода (экрана) в файл является стандартным для всех команд, выполняемых командным интерпретатором. Вы можете получить файл, содержащий список всех файлов текущей директории, если выполните команду ls-a с перенаправлением выходных данных:

\$ *ls -a > новый_файл*

Если имена входных файлов для команды cat не заданы, то она будет использовать в качестве входных данных информацию, которая вводится с клавиатуры, до тех пор, пока вы не наберете признак окончания ввода – комбинацию клавиш <CTRL> и <d>.

Таким образом, команда

\$ *cat > новый_файл* позволяет создать новый текстовый файл с именем «новый_файл» и содержимым, которое пользователь введет с клавиатуры.

У команды cat существует множество различных опций. Посмотреть ее полное описание можно в UNIX Manual.

Заметим, что наряду с перенаправлением выходных данных существует другой способ перенаправить входные данные. Если во время выполнения некоторой команды требуется ввести данные с

клавиатуры, можно положить их заранее в файл, а затем перенаправить стандартный ввод этой команды с помощью знака «меньше» – «<» – и следующего за ним имени файла с входными данными.

1.8 Шаблоны имен файлов

Шаблоны имен файлов могут применяться в качестве параметра для задания набора имен файлов во многих командах операционной системы. При использовании шаблона просматривается вся совокупность имен файлов, находящихся в файловой системе, и те имена, которые удовлетворяют шаблону, включаются в набор. В общем случае шаблоны могут задаваться с использованием следующих метасимволов:

- * – соответствует всем цепочкам литер, включая пустую;
- ? – соответствует всем одиночным литерам;
- [...] – соответствует любой литере, заключенной в скобки.

Пара литер, разделенных знаком минус, задает диапазон литер.

Так, например, шаблону *.c удовлетворяют все файлы текущей директории, чьи имена заканчиваются на .c. Шаблону [a-d]*удовлетворяют все файлы текущей директории, чьи имена начинаются с букв a, b, c, d. Существует одно ограничение на использование метасимвола * в начале имени файла, например, в случае шаблона *.c. Для таких шаблонов имена файлов, начинающиеся с символа точка, считаются не соответствующими шаблону.

1.9 Простейшие команды работы с файлами – cp, rm, mkdir, mv

Для нормальной работы с файлами необходимо не только уметь создавать файлы, просматривать их содержимое и перемещаться по логическому дереву файловой системы. Нужно уметь создавать собственные поддиректории, копировать и удалять файлы, переименовывать их. Это минимальный набор операций, не владея которым, нельзя чувствовать себя уверенно при работе с компьютером.

Для создания новой поддиректории используется команда **mkdir** (сокращение от **make directory**). В простейшем виде команда выглядит следующим образом:

\$ **mkdir имя_директории**, где «имя_директории» – полное или относительное имя создаваемой директории. У команды **mkdir** имеется набор опций, описание которых можно просмотреть с помощью утилиты **man**.

Для копирования файлов и директорий применяется команда **cp**. Данная команда может применяться в следующих формах.

\$ **cp файл_источник файл_назначения** - служит для копирования одного файла с именем «файл_источник» в файл с именем «файл_назначения».

Команда **cp** в форме

\$ **cp файл1 файл2 ... файлN дир_назначения** - служит для копирования файла или файлов с именами «файл1», «файл2», ... «файлN» в уже существующую директорию с именем «дир_назначения» под своими именами. Вместо имен копируемых файлов могут использоваться их шаблоны.

\$ **cp -r дир_источник дир_назначения** - служит для рекурсивного копирования одной директории с именем «дир_источник» в новую директорию с именем «дир_назначения». Если директория «дир_назначения» уже существует, то мы получаем команду **cp** в следующей форме

\$ **cp -r дир1 дир2 ... дирN дир_назначения** - служит для рекурсивного копирования директории или директорий с именами «дир1», «дир2», ... «дирN» в уже существующую директорию с именем «дир_назначения» под своими собственными именами. Вместо имен копируемых директорий могут использоваться их шаблоны.

Для удаления файлов или директорий применяется команда **rm** (сокращение от **remove**). Если вы хотите удалить один или несколько регулярных файлов, то простейший вид команды **rm** будет выглядеть следующим образом:

\$ **rm файл1 файл2 ... файлN**, где «файл1», «файл2», ... «файлN» – полные или относительные имена регулярных файлов, которые требуется удалить. Вместо имен файлов могут использоваться их шаблоны. Если вы хотите удалить одну или несколько директорий

вместе с их содержимым (рекурсивное удаление), то к команде добавляется опция -r:

\$ **rm -r дир1 дир2 ... дирN**, где «дир1», «дир2», ... «дирN» – полные или относительные имена директорий, которые нужно удалить. Вместо непосредственно имен директорий также могут использоваться их шаблоны.

У команды **rm** есть еще набор полезных опций, которые описаны в UNIXManual и могут быть просмотрены с помощью команды **man**. Командой удаления файлов и директорий следует пользоваться с осторожностью. Удаленную информацию восстановить невозможно. Если вы системный администратор и ваша текущая директория – это корневая директория, пожалуйста, не выполняйте команду **rm -r ***!

Для перемещения файлов и директорий используется команда mv (сокращение от move). Данная команда может применяться в следующих формах:

\$ **mv имя_источника имя_назначения** - для переименования или перемещения одного файла (неважно, регулярного или директории) с именем «имя_источника» в файл с именем «имя_назначения». При этом перед выполнением команды файла с именем «имя_назначения» существовать не должно.

\$ **mv имя1 имя2 ... имяN дир_назначения** - служит для перемещения файла или файлов (неважно, регулярных файлов или директорий) с именами «имя1», «имя2», ... «имяN» в уже существующую директорию с именем «дир_назначения» под собственными именами. Вместо имен перемещаемых файлов могут использоваться их шаблоны.

1.10 Пользователь и группа. Команды **chown и **chgrp**. Права доступа к файлу**

Как уже говорилось, для входа в операционную систему UNIX каждый пользователь должен быть зарегистрирован в ней под определенным именем. Вычислительные системы не умеют оперировать именами, поэтому каждому имени пользователя в системе соответствует некоторое числовое значение – его идентификатор UID (user identificator).

Все пользователи в системе делятся на группы. Например,

студенты одной учебной группы могут составлять отдельную группу пользователей. Группы пользователей также получают свои имена и соответствующие идентификационные номера – GID (group identifier). В одних версиях UNIX каждый пользователь может входить только в одну группу, в других – в несколько групп.

Команда `chown` предназначена для изменения собственника (хозяина) файлов. Нового собственника файла могут назначить только предыдущий собственник файла или системный администратор.

\$ `chown owner файл1 файл2 ... файлN` - параметр `owner` задает нового собственника файла в символьном виде, как его `username`, или в числовом виде, как его `UID`. Параметры «файл1», «файл2», ... «файлN» – это имена файлов, для которых производится изменение собственника. Вместо имен могут использоваться их шаблоны.

Для каждого файла, созданного в файловой системе, запоминаются имена его хозяина и группы хозяев. Заметим, что группа хозяев не обязательно должна быть группой, в которую входит хозяин. Упрощенно можно считать, что в операционной системе Linux при создании файла его хозяином становится пользователь, создавший файл, а его группой хозяев – группа, к которой этот пользователь принадлежит. Впоследствии хозяин файла или системный администратор могут передать его в собственность другому пользователю или изменить его группу хозяев с помощью команд `chown` и `chgrp`.

Команда `chgrp` предназначена для изменения группы собственников (хозяев) файлов.

\$ `chgrp group файл1 файл2 ... файлN` - новую группу собственников файла могут назначить только собственник файла или системный администратор. Параметр `group` задает новую группу собственников файла в символьном виде, как имя группы, или в числовом виде, как ее `GID`. Параметры «файл1», «файл2», ... «файлN» – это имена файлов, для которых производится изменение группы собственников. Вместо имен могут использоваться их шаблоны.

Для каждого файла выделяется три категории пользователей:

- пользователь, являющийся хозяином файла;
- пользователи, относящиеся к группе хозяев файла;
- все остальные пользователи.

Для каждой из этих категорий хозяин файла может определить различные права доступа к файлу. *Различают три вида прав доступа: право на чтение файла – r (от слова read), право на модификацию файла – w (от слова write) и право на исполнение файла — x (от слова execute).*

Команда **chmod** предназначена для изменения прав доступа к одному или нескольким файлам.

\$ chmod [who] { + / - | = } [perm] файл1 файл2 ... файлN - права доступа к файлу могут менять только собственник (хозяин) файла или системный администратор.

Параметр *who* определяет, для каких категорий пользователей устанавливаются права доступа. Он может представлять собой один или несколько символов:

a – установка прав доступа для всех категорий пользователей. Если параметр *who* не задан, то по умолчанию применяется *a*. При определении прав доступа с этим значением заданные права устанавливаются с учетом значения маски создания файлов;

u – установка прав доступа для собственника файла;

g – установка прав доступа для пользователей, входящих в группу собственников файла;

O – установка прав доступа для всех остальных пользователей.

Операция, выполняемая над правами доступа для заданной категории пользователей, определяется одним из следующих символов:

+ – добавление прав доступа;

- – отмена прав доступа;

= – замена прав доступа, т.е. отмена всех существовавших и добавление перечисленных.

Если параметр *perm* не определен, то все существовавшие права доступа отменяются.

Параметр *perm* определяет права доступа, которые будут добавлены, отменены или установлены взамен соответствующей командой. Он представляет собой комбинацию следующих символов или один из них:

r – право на чтение;
w – право на модификацию;
x – право на исполнение.

Параметры *файл1, файл2, ... файлN* – это имена файлов, для которых производится изменение прав доступа. Вместо имен могут использоваться их шаблоны.

Хозяин файла может изменять права доступа к нему, пользуясь командой `chmod`.

1.11 Вход удаленным пользователем

Для входа удаленным пользователем в систему UNIX используется утилита `ssh` (security shell). Для доступа к другим UNIX системам с UNIX машины

\$ ssh -l <Имя пользователя> <IP адрес удаленной машины>

Пользователь может набрать команду

\$ ssh -l <Имя пользователя> localhost для доступа по `ssh` к «своей» (локальной) машине.

1.12 Команды `write` и `wall`

Очень часто устанавливают многопользовательскую поддержку, многие люди работают на том же сервере через различные удаленные доступные рабочие варианты. Однако все эти пользователи системы Linux могут работать на соответствующем проекте или в команде, и даже если они не связаны, они принадлежат к одному рабочему месту. Таким образом, вполне вероятно, им придется общаться в любой момент времени. С утилитой Linux **write**, Linux пользователи имеют удобный способ общения друг с другом. Можно послать сообщение любому другому пользователю и разрешить войти в Linux машину. Более того, можно посылать сообщения любому пользователю в той же сети, даже на другой машине хозяина.

Синтаксис выглядит так: **\$ write person** Здесь **person** - имя пользователя, если он находится на той же машине, машины или `username@hostname` в случае, если пользователь принадлежит к другой машине хозяина и необходима идентификация, когда один пользователь заходит более чем один раз. Рассмотрим, как передается сообщения среди пользователей на той же машине.

\$ w -s — данная команда выводит всех активных пользователей системы;

\$write person — вместо **person** мы можем вписать имя любого активного пользователя. После нажатия клавиши ввода появляется возможность писать наше сообщение.

Команда **wall** используется для передачи сообщения всем пользователям системы. Однако для получения этого сообщения пользователи должны установить разрешение их **mesg** на "да". В использовании довольно проста, что легко понять на следующем примере;

\$ wall - как только ввод сообщения завершен, нужно нажать комбинацию клавиш **<CTRL> + <D>**.

После этого все пользователи получают сообщение.

Таблица 1.1 - Основные информационные команды

Команды	Описание
<i>hostname</i>	Вывести или изменить сетевое имя машины.
<i>whoami</i>	Вывести имя, под которым я зарегистрирован.
<i>date</i>	Вывести или изменить дату и время.
<i>time</i>	Получить информацию о времени, нужном для выполнения процесса
<i>who</i>	Определить, кто из пользователей работает на машине.
<i>rwho -a</i>	Определение всех пользователей, подключившихся к вашей сети. Для выполнения этой команды требуется, чтобы был запущен процесс rwho . Если такого нет - запустите "setup" под суперпользователем.
<i>finger</i> <i>[имя_пользователя]</i>	Системная информация о зарегистрированном пользователе. Попробуйте: finger root
<i>ps -a</i>	Список текущих процессов
<i>df -h</i>	(=место на диске) Вывести информацию о свободном и используемом месте на дисках (в читабельном виде).

<i>Arch или uname -m uname -r</i>	отобразить архитектуру компьютера отобразить используемую версию ядра
<i>find / -name file1 find / -user user1</i>	найти файлы и директории с именем file1. Поиск начать с корня (/) найти файл и директорию, принадлежащие пользователю user1. Поиск начать с корня (/)
<i>top</i>	отобразить запущенные процессы, используемые ими ресурсы и другую полезную информацию (с автоматическим обновлением данных)
<i>Kill -9 98989 или kill -KILL 98989</i>	«убить» процесс с PID 98989 «на смерть» (без соблюдения целостности данных)

1.13 Командный интерпретатор Shell

Командный интерпретатор в среде UNIX выполняет две основные функции:

- представляет интерактивный интерфейс с пользователем, т. е. выдает приглашение и обрабатывает вводимые пользователем команды;

- обрабатывает и исполняет текстовые файлы, содержащие команды интерпретатора (командные файлы).

В последнем случае операционная система позволяет рассматривать командные файлы как разновидность исполняемых файлов. Соответственно, различают два режима работы интерпретатора: интерактивный и командный.

Существует несколько типов оболочек в мире UNIX. Две главные - это «Bourne shell» и «C shell». Bourne shell (или просто shell) использует командный синтаксис, похожий на первоначальный для UNIX. В большинстве UNIX-систем Bourne shell имеет имя /bin/sh (где sh сокращение от «shell»). C shell используется иной синтаксис, чем-то напоминающий синтаксис языка программирования Си. В большинстве UNIX-систем он имеет имя /bin/csh.

В Linux есть несколько вариаций этих оболочек. Две наиболее часто используемые – это Новый Bourne shell (Bourne Again Shell) или

«Bash» (/bin/bash) и Tcsh (/bin/tcsh). Bash - это развитие прежнего shell с добавлением многих полезных возможностей, частично содержащихся в C shell.

Поскольку Bash можно рассматривать как надмножество синтаксиса прежнего shell, любая программа, написанная на sh shell должна работать и в Bash. Tcsh является расширенной версией C shell.

При входе в систему пользователю загружается командный интерпретатор по умолчанию. Информация о том, какой интерпретатор использовать для конкретного пользователя, находится в файле /etc/passwd.

Настройка Shell. Файлы инициализации, используемые в bash: /etc/profile (устанавливается системным администратором, выполняется всеми экземплярами начальных пользовательских bash, вызванными при входе пользователей в систему), \$HOME/.bash_profile (выполняется при входе пользователя) и \$HOME/.bashrc (выполняемый всеми прочими не начальными экземплярами bash). Если .bash_profile отсутствует, вместо него используется .profile. Переменная HOME указывает на домашний каталог пользователя. tcsh использует следующие сценарии инициализации: /etc/csh.login (выполняется всеми пользовательскими tcsh в момент входа в систему), \$HOME/.tcshrc (выполняется во время входа в систему и всеми новыми экземплярами tcsh) и \$HOME/.login (выполняется во время входа после .tcshrc). Если .tcshrc отсутствует, вместо него используется .cshrc.

Командные файлы. Командный файл в UNIX представляет собой обычный текстовый файл, содержащий набор команд UNIX и команд Shell.

Для того чтобы командный интерпретатор воспринимал этот текстовый файл как командный, необходимо установить атрибут на исполнение.

Установку атрибута на исполнение можно осуществить командой `chmod` или через `ts` по клавише F9 выйти в меню и выбрать вкладку File, далее выбрать изменение атрибутов файла.

Например.

```
$ echo "ps -af" > commandfile  
$ chmod +x commandfile  
$ ./commandfile
```

В представленном примере команда `echo "ps -af" > commandfile` создаст файл с одной строкой `"ps -af"`, команда `chmod +x commandfile` установит атрибут на исполнение для этого файла, команда `./commandfile` осуществит запуск этого файла.

Переменные shell. Имя shell-переменной - это начинающаяся с буквы последовательность букв, цифр и подчеркиваний. Значение shell-переменной - строка символов.

Например: `Var = "String"` или `Var = String`

Команда **`echo $Var`** выведет на экран содержимое переменной `Var`, т. е. строку `'String'`, на то, что мы выводим содержимое переменной, указывает символ `'$'`.

Так, команда `echo Var` выведет на экран просто строку `'Var'`.

Еще один вариант присвоения значения переменной **`Var = `набор команд UNIX``**

Обратные кавычки говорят о том, что сначала должна быть выполнена заключенная в них команда, а результат ее выполнения, вместо выдачи на стандартный выход, приписывается в качестве значения переменной.

`CurrentDate = `date`` - Переменной `CurrentDate` будет присвоен результат выполнения команды `date`.

Можно присвоить значение переменной и с помощью команды `«read»`, которая обеспечивает прием значения переменной с (клавиатуры) дисплея в диалоговом режиме.

Например:

```
echo "Введите число"  
read X1  
echo "вы ввели -" $X1
```

Несмотря на то что shell-переменные в общем случае воспринимаются как строки, т. е. `«35»` - это не число, а строка из двух символов `«3»` и `«5»`, в ряде случаев они могут интерпретироваться иначе, например, как целые числа.

Разнообразные возможности имеет команда `"expr"`.

Например, командный файл:

```
x=7
```

```
y=2
```

```
rez=`expr $x + $y`
```

```
echo результат=$rez
```

выдаст на экран результат=9

Порядок выполнения работы:

1. Объясните основные моменты работы с системой UNIX в терминальном режиме: вход в систему обычным символьным терминалом, переключение между терминалами; регистрация удаленных терминалов с помощью протокола ssh; запуск утилиты mc; получение информации о пользователях, зарегистрированных в системе (команды who, w, finger).
2. Объясните организацию структуры каталогов в UNIX, рассмотрите основные каталоги /etc, /bin, /usr, /proc, их назначение.
3. Рассмотрите основные информационные команды и команды управления процессами.
4. Рассмотрите настройку shell (bash) и переменных среды окружения.
5. Рассмотрите основы написания сценариев на языке shell, изучите основные команды языка shell (bash).
6. Напишите свой собственный сценарий на языке shell с использованием изученных команд.
7. Получите подробную информацию о файлах домашней директории.
8. Создайте новый файл и посмотрите на права доступа к нему, установленные системой при его создании.
9. Убедитесь, что вы находитесь в своей домашней директории, и создайте новый текстовый файл. Введите туда информацию: ФИО студента, № группы. Скопируйте этот файл в другую директорию.
10. Реализуйте командный файл, который выводит: дату, системную информацию и текущего пользователя.

Варианты индивидуальных заданий

1. Написать командный файл, реализующий меню из трех пунктов: 1-й пункт: ввести пользователя и вывести на экран все процессы, запущенные данным пользователем; 2-й пункт: показать всех пользователей, в настоящий момент находящихся в системе; 3-й пункт: завершение.
2. Написать командный файл, реализующий меню из трех пунктов: 1-ый пункт: вывести всех пользователей, в настоящее время, работающих в системе; 2-й пункт: послать сообщение пользователю, имя пользователя, терминал и сообщение вводятся с клавиатуры; 3-ий пункт: завершение.
3. Написать командный файл, реализующий меню из трех пунктов: 1-й пункт: показать все процессы пользователя, запустившего данный командный файл; 2-й пункт: послать сигнал завершения процессу текущего пользователя (ввести PID процесса); 3-й пункт: завершение.
4. Написать командный файл, подсчитывающий количество активных терминалов пользователя (имя пользователя вводится с клавиатуры).
5. Написать командный файл, посылающий сообщение всем активным пользователям (сообщение находится в файле).
6. Написать командный файл, посылающий сигнал завершения процессам текущего пользователя. Символьная маска имени процесса вводится с клавиатуры.
7. Написать командный файл, подсчитывающий количество определенных процессов пользователя (ввести имя пользователя и название процесса).
8. Реализовать Меню из двух пунктов: 1-й пункт: определить количество запущенных данным пользователем процессов `bash` (предусмотреть ввод имени пользователя); 2-й пункт: завершить все процессы `bash` данного пользователя.
9. Реализовать Меню из трех пунктов: 1-й пункт: поиск файла в каталоге <Имя файла> и <Имя каталога> вводятся пользователем; 2-й пункт: копирование одного файла в другой каталог - <Имя файла> и <Имя каталога> вводятся; 3-й пункт: завершение командного файла.
10. Написать командный файл, который в цикле по нажатию клавиши выводит информацию о системе, активных пользователях в системе, а для введенного имени пользователя выводит список активных процессов данного пользователя.
11. Реализовать командный файл, который при старте выводит

информацию о системе, информацию о пользователе, запустившем данный командный файл, далее в цикле выводит список активных пользователей в системе – запрашивает имя пользователя и выводит список всех процессов `bash`, запущенных данным пользователем.

12. Реализовать командный файл, позволяющий в цикле посылать всем активным пользователям сообщение – сообщение вводится с клавиатуры. Командный файл при старте выводит имя компьютера, имя запустившего командный файл пользователя, тип операционной системы, IP-адрес машины.

13. Реализовать командный файл, позволяющий в цикле посылать всем активным пользователям (исключая пользователя, запустившего данный командный файл) сообщение – сообщение вводится с клавиатуры. Командный файл при старте выводит имя компьютера, имя запустившего командный файл пользователя, тип операционной системы, список загруженных модулей.

14. Реализовать командный файл, который при старте выводит информацию о системе, информацию о пользователе, запустившем данный командный файл, далее в цикле выводит список активных пользователей в системе: запрашивает имя пользователя и выводит список всех терминалов, на которых зарегистрирован этот пользователь.

15. Реализовать командный файл, который выводит следующие данные: дату, информацию о системе, текущий каталог, текущего пользователя, настройки домашнего каталога текущего пользователя. Далее в цикле выводит список активных пользователей: запрашивает имя пользователя и выводит информацию об активности данного пользователя.

16. Реализовать командный файл, который выводит: дату в формате «день – месяц – год – время», информацию о системе в формате: «имя компьютера : версия ОС : IP адрес : имя текущего пользователя : текущий каталог». Выводит настройки домашнего каталога текущего пользователя и основные переменные окружения. Далее в цикле выводит список активных пользователей: запрашивает имя пользователя и выводит информацию об активности введенного пользователя.

17. Реализовать командный файл, реализующий символьное меню (в цикле). 1-й пункт: вывод полной информации о файлах каталога. Ввести имя каталога для отображения. 2-й пункт: изменить атрибуты

файла. Файл вводится с клавиатуры по запросу, атрибуты, которые требуется установить, также вводятся. После изменения атрибутов вывести на экран расширенный список файлов для проверки установленных атрибутов 3-й пункт: выход. При старте командный файл выводит информацию об имени компьютера, IP- адреса и список всех пользователей, зарегистрированных в данный момент на компьютере.

18. Реализовать командный файл, реализующий символьное меню (в цикле). 1-й пункт: вывод полной информации о файлах каталога. Ввести имя каталога для отображения. 2-й пункт: создать командный файл. Файл вводится с клавиатуры по запросу, далее изменяется атрибут файла на исполнение, затем вводится с клавиатуры строка, которую будет исполнять командный файл. После изменения атрибутов вывести на экран расширенный список файлов для проверки установленных атрибутов и запустить созданный командный файл. 3-й пункт: выход. При старте командный файл выводит информацию об имени компьютера, IP- адреса и список всех пользователей, зарегистрированных в данный момент на компьютере.

19. Написать командный файл, реализующий символьное меню. 1-й пункт: работа с информационными командами (реализовать все основные информационные команды). 2-й пункт: копирование файлов. В этом пункте выводится информация о содержимом текущего каталога, далее предлагается интерфейс копирования файла: ввод имени файла и ввод каталога для копирования. По выполнению пункта выводится содержимое каталога, куда был скопирован файл, и выводится содержимое скопированного файла. 3-й пункт: выход.

Контрольные вопросы

1. Опишите полные и относительные имена файлов.
2. Понятие о текущей и домашней директории. Команда `pwd`.
3. Опишите использование команд `cd` и `ls`.
4. Команда `cat`, примеры использования.
5. Команды работы с файлами (`cp`, `rm`, `mkdir`, `mv`).
6. Система `ms`.
7. Команды `chown` и `chgrp`. Права доступа к файлу.
8. Использование команд `chmod` и `umask`.
9. Командный интерпретатор `shell`.

Лабораторная работа 2 **АДМИНИСТРИРОВАНИЕ**

Цели и задачи

Рассмотреть функции администратора и суперпользователей. Научиться устанавливать программы различными способами. Рассмотреть различные способы архивирования.

2.1 Администратор и суперпользователь

Суперпользователь

Во всех системах на базе Linux всегда есть один привилегированный пользователь, который называется root или суперпользователь. Полномочия этого пользователя не ограничены ничем, он может делать в системе абсолютно все, что угодно. Кроме того, большинство системных процессов работают от имени root. Обычный же пользователь в Linux не может устанавливать и удалять программы, управлять системными настройками и изменять файлы вне своего домашнего каталога. Поскольку использование суперпользователя крайне опасно, в UNIX-системах он скрыт внутри системы, а управлением занимаются обычные пользователи со специальными административными привилегиями. В суперпользователя можно превратиться. Для этого необходимо выполнить команду:

su # Super User

Администратор

Администратор по умолчанию может по запросу делать все то же самое, что и суперпользователь, однако случайно что-то испортить из-под администратора нельзя, т. к. перед выполнением каждого опасного действия система спрашивает у пользователя-администратора его пароль. Администратор является обычным пользователем, однако при необходимости он может вмешаться в работу системы, но для этого ему потребуется ввести свой пароль. Главное отличие администратора от суперпользователя заключается в необходимости вводить пароль для выполнения любого потенциально опасного действия.

2.2 Команда регистрации нового пользователя

Добавление пользователя осуществляется при помощи команды `useradd`. Пример использования:

sudo useradd <имя пользователя>

Эта команда создаст в системе нового пользователя. Чтобы изменить настройки создаваемого пользователя, вы можете использовать следующие ключи:

Таблица 2.1. Ключи для изменения настроек создаваемого пользователя

Ключ	Описание
-b	Базовый каталог. Это каталог, в котором будет создана домашняя папка пользователя. По умолчанию <code>/home</code>
-c	Комментарий. В нем вы можете напечатать любой текст.
-d	Название домашнего каталога. По умолчанию название совпадает с именем создаваемого пользователя.
-e	Дата, после которой пользователь будет отключен. Задается в формате ГГГГ-ММ-ДД. По умолчанию отключено.
-f	Количество дней, которые должны пройти после устаревания пароля до блокировки пользователя, если пароль не будет изменен (период неактивности). Если значение равно 0, то запись блокируется сразу после устаревания пароля, при -1 - не блокируется. По умолчанию -1.
-g	Первичная группа пользователя. Можно указывать как <code>GID</code> , так и имя группы. Если параметр не задан, будет создана новая группа, название которой совпадает с именем пользователя.
-G	Список вторичных групп, в которых будет находиться создаваемый пользователь.
-k	Каталог шаблонов. Файлы и папки из этого каталога будут помещены в домашнюю папку пользователя. По умолчанию <code>/etc/skel</code> .
-m	Ключ, указывающий, что необходимо создать домашнюю папку. По умолчанию домашняя папка не создается .

-p	Зашифрованный пароль пользователя. По умолчанию пароль не задается, но пользователь будет заблокирован до установки пароля
-s	Оболочка, используемая пользователем. По умолчанию /bin/sh.
-u	Вручную задать UID пользователю.

2.3 Установка программ

В UNIX-системах, как и в других операционных системах, есть понятие зависимостей. Это значит, что программу можно установить, только если уже установлены пакеты, от которых она зависит. Такая схема позволяет избежать дублирования данных в пакетах (например, если несколько программ зависят от одной и той же библиотеки, то она установится один раз отдельным пакетом). В отличие от, например, Slackware или Windows, в UNIX зависимости разрешаются пакетным менеджером (Synaptic, apt, Центр приложений, apt-get, aptitude) – он автоматически установит зависимости из репозитория. Зависимости придется устанавливать вручную, если нужный репозиторий не подключен, недоступен, если нужного пакета нет в репозитории, если вы устанавливаете пакеты без использования пакетного менеджера (используете Gdebi или dpkg), если вы устанавливаете программу не из пакета (компилируете из исходников, запускаете установочный run/sh скрипт).

Установка из репозитория

Репозиторий – место централизованного хранения пакетов программного обеспечения. Использование репозитория позволяет упростить установку программ и обновление системы. Пользователь волен выбирать, какими репозиториями будет пользоваться, и может создать собственный. Список используемых репозиториях содержится в файле /etc/apt/sources.list и в файлах каталога/etc/apt/sources.list.d/, проще всего его посмотреть через специальное приложение, которое можно вызвать через главное меню: Система → Администрирование

→ *Источники Приложений*, или через Менеджер пакетов Synaptic.

Если не добавлялись локальные репозитории (например, CD/DVD диски), то для установки программ из репозитория необходим будет интернет.

У такого метода установки программ есть масса преимуществ: это удобно, устанавливаются уже протестированные программы, которые гарантированно будут работать, зависимости между пакетами будут решаться автоматически, информирование о появлении в репозитории новых версий установленных программ.

С использованием графического интерфейса

Выберите Система → Администрирование → Менеджер пакетов Synaptic и получите более функциональный инструмент для работы с пакетами. В частности можно например устанавливать программы частично. Запустите программу Менеджер пакетов Synaptic: Система → Менеджер пакетов Synaptic. По запросу введите свой пароль. В запустившейся программе нажмите кнопку «Обновить», подождите, пока система обновит данные о доступных программах.

В списке доступных программ сделайте двойной клик на нужной программе (либо клик правой кнопкой - пункт «Отметить для установки»). После того, как все нужные программы помечены для установки, нажмите кнопку «Применить». Подождите, пока необходимые пакеты будут скачаны и установлены. Схожие функции выполняет программа «Установка и удаление приложений», ее можно легко найти в меню Приложения → Установка/удаление...

С использованием командной строки

Установка из командной строки позволяет получить больше информации о процессе установки и позволяет гибко его настраивать, хотя и может показаться неудобной начинающему пользователю.

Обновить данные о доступных в репозиториях программах можно командой:

sudo apt-get update

Для установки нужной программы:

sudo apt-get install имя-программы

Например:

sudo apt-get install libsexymm2

Если нужно установить несколько программ, то их можно перечислить через пробел, например:

sudo apt-get install libsexymm2 nmap

Если потребуется - ответьте на задаваемые вопросы (для положительного ответа нужно ввести Y или D). Программа будет установлена, если она уже установлена - она будет обновлена.

Для поиска программы в списке доступных пакетов:

sudo apt-cache search keyword

где **keyword** - название программы, часть названия программы или слово из ее описания.

Установка из deb-пакета

Если нужной программы нет в основном репозитории и у автора программы нет своего репозитория, либо если репозитории недоступны (например, нет интернета), то программу можно установить из deb-пакета (скачанного заранее/принесенного на USB накопителе). Если deb-пакет есть в официальном репозитории, то его можно скачать с сайта <http://packages.ubuntu.com>. Часто deb-пакет можно скачать с сайта самой программы. Можно также воспользоваться поиском на сайте <http://getdeb.net>. Минус такого подхода – менеджер обновлений не будет отслеживать появление новых версий установленной программы.

С использованием графического интерфейса

Перейдите в папку, где находится deb-пакет, откройте свойства файла (правая клавиша → Свойства), во вкладке «Права» разрешите выполнение файла (галочка у «Разрешить исполнение файла как программы»). Далее закрываем свойства файла, и по двойному щелчку Nautilus предложит нам открыть код или выполнить файл. Запускаем. Либо возможно это сделать специальным установщиком GDebi. Установить можно из Центра приложений, вписав в поиск GDebi, либо вписав в командную строку:

sudo apt-get install GDebi

После установки запускаем deb-пакет с помощью установщика программ GDebi; все, что от вас потребуется – это просто нажать кнопку «Установить пакет».

Возможные ошибки

Пакет не может быть установлен. Например, он предназначен для другой архитектуры.

В системе отсутствуют необходимые устанавливаемому приложению пакеты. В таком случае «Установщик программ GDebi» автоматически попытается получить нужные пакеты из репозитория. Или же вы можете самостоятельно скачать требуемые пакеты и установить их.

С использованием командной строки

Установка выполняется с помощью программы dpkg:

sudo dpkg -i /home/user/soft/ntlmaps_0.9.9.0.1-10_all.deb

При использовании dpkg нужно ввести полное имя файла (а не только название программы). Можно одной командой установить сразу несколько пакетов, например, следующая команда установит все deb-пакеты в директории:

sudo dpkg -i /home/user/soft/ntlmaps_*.deb

Это бывает полезно для установки пакета программы вместе с пакетами зависимостей.

Установка программ с собственным инсталлятором из файлов sh, run

Иногда программы могут распространяться с собственным инсталлятором. Это ничем не отличается от ситуации в Windows. Только здесь, распаковав **tar.gz** архив с дистрибутивом программы, вы вместо **setup.exe** увидите что-то наподобие **install.sh**. Это заранее собранный пакет ПО, он берет на себя работу по размещению файлов в нужных местах и прописыванию нужных параметров. При этом пропадает возможность управлять таким ПО с помощью пакетного менеджера. Пользоваться такими пакетами нежелательно, но если выбора нет, то переходим в директорию с файлом, например:
`cd ~/soft`

Разрешение выполнять этот файл:

chmod +x install.sh

Запуск файла:

sudo ./install.sh

Иногда программу можно установить и без прав суперпользователя (без `sudo`), но это, скорее, исключение.

Иногда дистрибутив программы распространяется в виде самораспаковывающегося архива. В таком случае это будет просто один единственный файл `.sh`, который и нужно запустить. Дальше нужно будет ответить на ряд вопросов, как это делается в Windows. Так устанавливаются официальные драйверы nVidia, ATI, среда разработчика NetBeans и т.п.

Есть программы, которые не нуждаются в инсталляции и распространяются в виде обычного архива `tar.gz`, который просто достаточно куда-то распаковать. В Windows тоже есть такие программы, их еще часто называют словом *Portable*. Такие программы обычно можно устанавливать в любой каталог, но обычно стандартное место – это каталог `/opt`. Конечно, пункты на запуск в меню придется добавлять вручную, для этого нужно щелкнуть правой кнопкой по заголовку меню «Программы» и выбрать «Правка меню».

Установка из исходных кодов

Если для системы нигде нет `deb`-пакетов, то программу можно скомпилировать самостоятельно из исходных кодов, которые можно скачать на официальном сайте любой Open Source программы либо из `source` – репозитория дистрибутива.

Основное, что понадобится – это средства для компиляции, поэтому сначала нужно установить пакет `build-essential`. Далее нужно распаковать архив с кодами программы в какую-то временную папку. Потом нужно найти файл `README` или `INSTALL`, прочитать его и выполнить то, что там написано. Чаше установка программ таким способом ограничивается последовательным выполнением следующих команд:

./configure
make
sudo make install

Но в некоторых случаях могут быть отличия. Кроме того, после выполнения скрипта *./configure* можно получить сообщение о том, что в системе не установлено библиотек, нужных для компиляции программы. В таком случае нужно будет установить их самостоятельно и повторить процесс. Обычно процесс компиляции занимает определенное время и напрямую зависит от мощности вашего компьютера.

Автоматическая установка зависимостей при сборке из исходников

Такой тип установки лучше, чем просто *./configure && make && make install*, и подходит для установки программ отсутствующих в репозиториях.

Установка *auto-apt*:

sudo apt-get install auto-apt

Переходим в папку с распакованными исходниками и командуем:

sudo auto-apt update && auto-apt -y run ./configure

Команда *auto-apt* сама доставит необходимые пакеты для сборки и уменьшит количество вопросов. Создание *deb* пакета для более простой работы в дальнейшем (установка, удаление, и прочее):

checkinstall -D

2.4 Архивирование. Копирование файлов на стример. Архиватор tar

tar – наиболее распространенный архиватор, используемый в Linux-системах. Сам по себе **tar** не является архиватором в привычном понимании этого слова, т. к. он самостоятельно не использует сжатие. В то же время многие архиваторы (например, **Gzip** или **bzip2**) не умеют сжимать несколько файлов, а работают только с одним файлом или входным потоком. Поэтому чаще всего эти программы используются вместе. **tar** создает несжатый архив, в который помещаются выбранные файлы и каталоги, при этом сохраняя

некоторые их атрибуты (такие, как права доступа). После этого полученный файл *.tar сжимается архиватором, например, gzip. Вот почему архивы обычно имеют расширение .tar.gz или .tar.bz2 (для архиваторов gzip и bzip2 соответственно).

Использование

tar запускается с обязательным указанием одного из основных действий, самые распространенные из которых – создание и распаковка архивов. Далее задаются прочие параметры, зависящие от конкретной операции.

Создание архива

Для создания архива нужно указать **tar** соответствующее действие, что делается с помощью ключа -с. Кроме того, для упаковки содержимого в файл необходим ключ -f. Далее указываются сначала имя будущего архива, а затем те файлы, которые необходимо упаковать.

tar -cf txt.tar *.txt

Эта команда упакует все файлы с расширением txt в архив txt.tar. Так и создается простейший архив без сжатия. Для использования сжатия не нужно запускать что-либо еще, достаточно указать **tar**, каким архиватором следует сжать архив. Для двух самых популярных архиваторов gzip и bzip2 ключи будут -z и -j соответственно.

tar -cvzf files.tar.gz ~/files

упакует папку ~/files со всем содержимым в сжатый с помощью gzip архив.

tar -cvjf files.tar.bz2 ~/files

создаст аналогичный архив, используя для сжатия bzip2. Ключ -v включает вывод списка упакованных файлов в процессе работы. Помимо gzip и bzip2 можно использовать, например, lzma (ключ -lzma) или xz (ключ -J), при этом соответствующий архиватор должен быть установлен в системе.

Распаковка архива

Действие «распаковка» задается с помощью ключа -x. И тут снова потребуется ключ -f для указания имени файла архива. Также

необходим ключ `-v` для визуального отображения хода процесса.

tar -xvf /path/to/archive.tar.bz2

распакует содержимое архива в текущую папку. Альтернативное место для распаковки можно указать с помощью ключа `-C`:

tar -xvf archive.tar.bz2 -C /path/to/folder

Просмотр содержимого архива

Для просмотра содержимого архива используется следующая команда:

tar -tf archive.tar.gz

Она выведет простой список файлов и каталогов в архиве. Если же добавить ключ `-v`, будет выведен подробный список с указанием размера, прав доступа и прочих параметров (так же, как по `ls -l`)

Прочие возможности

tar предоставляет множество полезных возможностей. Например, можно указать файлы и каталоги, которые не будут включены в архив, добавить файлы в существующий архив, взять список объектов для запаковки из текстового файла и многое другое. Во всем многообразии опций, как всегда, поможет разобраться

man tar

или же

tar --help

2.5 Копирование файлов на стример. Команда CPIO (Copy In/Out)

Команда `cpio -o` берет с системного ввода список имен и склеивает эти файлы вместе в один архив, выталкивая его на свой системный вывод.

Сбросить на ленту файлы по списку: ***o*** - (output) – создавать архив.

odc – записывать в «совместимом формате» (чтобы архив можно было считать на Besta или Sun)с – записывать в «престарелом» совместимом формате.

cat spisok | cpio -ovB -H odc > /dev/rmt/ctape1

find katalog -print | cpio -ovc > arhiwnyj-fajl.cpio Команда `cpio -i` читает с системного ввода `cpio`-архив и извлекает из него файлы.

Просмотреть содержание стримера.

cpio -itB < /dev/rmt/ctape

Извлечь файлы со стримера.

cpio -idmvB ["шаблон" ...] < /dev/rmt/ctape

-**B** – размер блока 5120 байт - стримерный формат.

-**d** – создавать каталоги в случае необходимости.

-**v** – вывести список имен обработанных файлов.

-**m** – сохранять прежнее время последней модификации.

-**f** – брать все файлы, кроме указанного шаблоном.

-**u** – безусловно заменять существующий файл архивным.

-**l** – где можно, не копировать, а делать ссылки.

2.6 Архивация со сжатием

Архиваторы tar и cpio, в отличие от DOS-архиваторов, не занимаются компрессией. Чтобы получить сжатый архив, нужно воспользоваться специализированной командой **compress** или **gzip**. Команда **compress** читает свой системный ввод, а на свой системный выход подает «прожатые» данные. Команда **zcat** («сжатый cat») читает с системного входа «прожатый» файл, а на выход подает «разжатые» данные.

Создать сжатый tar-архив:

tar -cvf - emacs-19.28 | compress > emacs-19.28.tar.Z

Прочитать оглавление сжатого tar-архива:

zcat < emacs-19.28.tar.Z | tar -tvf -

Обратите внимание на ключ минус "-" на том месте, где в tar нужно указывать имя файла с архивом. Он означает «брать данные со стандартного входа» (или выводить архив на стандартный выход).

GNU Zip – достаточно известный упаковщик, имеет степень сжатия более высокую, чем у **compress**, почти как у **arj** или **pkzip**.

Создать сжатый cpio архив, используя «компрессор» **gzip**:

```
find . -print / cpio -ovcaB / gzip > arhiw.gz
```

Извлечь файлы из сжатого cpio-архива:

```
gunzip < arhiw.gz / cpio -idmv
```

Другие утилиты архивации

В зависимости от версии UNIX могут существовать и другие программы для бэкапирования и создания архивов.

backup/restore

dump

fbackup/frestore (HP/UX)

pac

...

2.7 Менеджер пакетов Synaptic

Менеджер пакетов Synaptic позволяет полностью управлять отдельными пакетами в системе. Основное его отличие от Центра приложений, кроме более богатого функционала, в том, что он работает на уровне пакетов, а не приложений (Приложение и пакет это не одно и то же. Каждое приложение состоит из одного или более пакетов)

Найти Synaptic можно в меню *Системные* → *Менеджер пакетов Synaptic*. Для запуска понадобится ввести свой административный пароль. При первом входе появляется краткая справка:

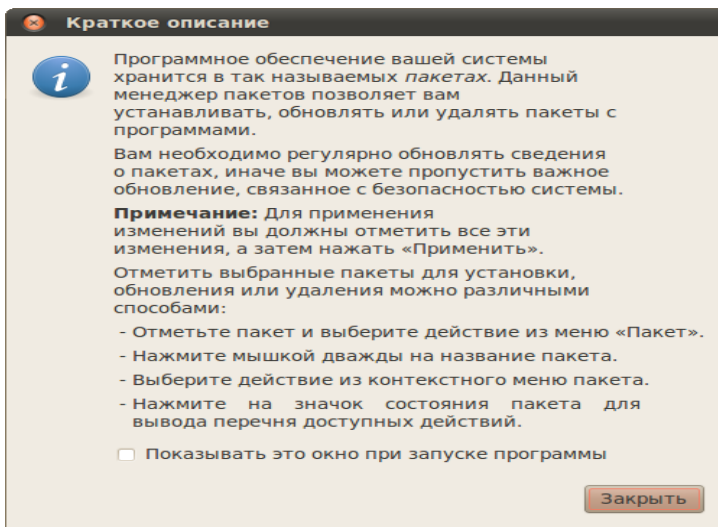


Рис. 2.1 Справка

Интерфейс Synaptic немного напоминает Центр приложений: слева находится колонка с категориями пакетов, под ней – переключатель способа сортировки по категориям, а справа находится собственно список пакетов и под ним описание текущего выбранного пакета:

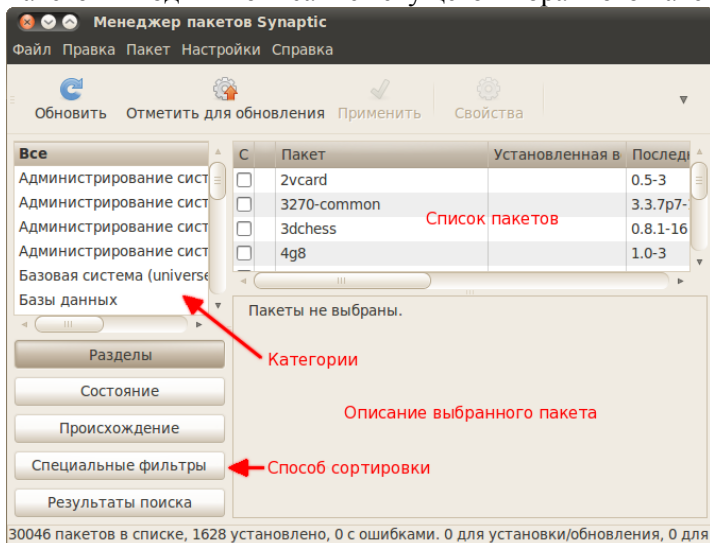


Рис. 2.2 Интерфейс менеджера пакетов Synaptic

Также на верхней панели есть строка поиска, а кроме нее кнопки, позволяющие совершать некоторые операции. При нажатии на кнопку «Обновить» будет произведено обновление индексов всех репозиторий, при нажатии на кнопку «Отметить для обновления» собственно будут отмечены для обновления все пакеты, для которых доступны новые версии, ну а кнопка «Применить» нужна для применения всех внесенных изменений.

Установленные пакеты помечаются зелеными квадратиками, а неустановленные – белыми. Изменить состояние того или иного пакета можно нажав правой кнопкой мыши на его название в списке и выбрав нужное действие:

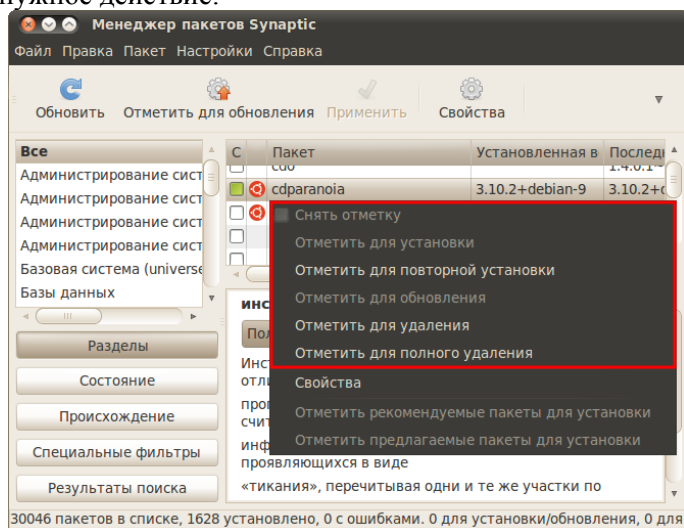


Рис. 2.3 Менеджер пакетов Synaptic. Изменение состояния пакетов

В отличие от Центра приложений внесенные через Synaptic изменения вступают в силу только после нажатия на кнопку «Применить» на панели инструментов.

Удалить пакет можно одним из двух способов: либо просто удалить файлы пакета, либо удалить их вместе со всеми пользовательскими настройками, относящимися к удаляемому пакету. Отличаются эти способы вот чем: многие программы создают в домашних папках пользователей файлы со своими настройками, а эти программы при простом удалении удаляются без пользовательских настроек, а при полном - с ними.

Synaptic, как и остальные инструменты управления пакетами, автоматически следит за разрешением всех зависимостей и ликвидацией различных конфликтов. Мало того, при совершении любых действий Synaptic выдаст вам окно с подробным описанием вносимых изменений.

2.8 Консольные инструменты управления пакетами

Утилита **dpkg**–

Существуют два основных инструмента работы с пакетами: **aptitude** и **dpkg**. **dpkg** это низкоуровневая программа управления пакетами, единственная полезная ее функция для обычного пользователя – это прямая установка пакета из deb-файла. Выполняется она командой

sudo dpkg -i имя_пакета.deb

Для того чтобы команда успешно выполнялась, в системе должны присутствовать все зависимости устанавливаемого пакета, поскольку **dpkg** не умеет их разрешать и скачивать что-либо из репозитория. Можно так же ставить несколько пакетов за раз, передавая их все как аргументы команде **dpkg -i**.

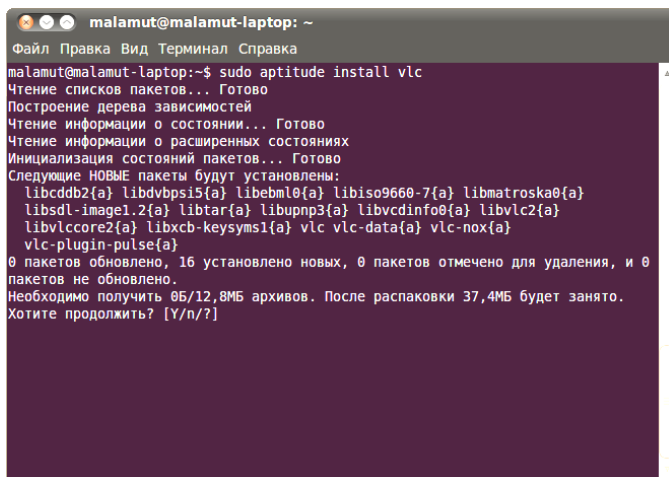
Утилита **aptitude**

Основным же консольным инструментом работы с пакетами является **aptitude**. В некотором смысле это консольный аналог менеджера пакетов Synaptic, хотя **aptitude** на самом деле обладает куда как большим функционалом.

Установить пакеты из репозитория можно командой

sudo aptitude install имя_пакета1 [имя_пакета2 ...]

Сколько бы ни было указано пакетов, **aptitude** автоматически разрешит все зависимости и предложит конечный вариант необходимых действий, останется только лишь согласиться, нажав Enter:



```
malamut@malamut-laptop: ~  
Файл Правка Вид Терминал Справка  
malamut@malamut-laptop:~$ sudo aptitude install vlc  
Чтение списков пакетов... Готово  
Построение дерева зависимостей  
Чтение информации о состоянии... Готово  
Чтение информации о расширенных состояниях  
Инициализация состояний пакетов... Готово  
Следующие НОВЫЕ пакеты будут установлены:  
  libcdt2{a} libdvbpsi5{a} libebml0{a} libiso9660-7{a} libmatroska0{a}  
  libSDL-image1.2{a} libtar{a} libunp3{a} libvcdinfo0{a} libvlc2{a}  
  libvlccore2{a} libxcb-keysyms1{a} vlc vlc-data{a} vlc-nox{a}  
  vlc-plugin-pulse{a}  
0 пакетов обновлено, 16 установлено новых, 0 пакетов отмечено для удаления, и 0  
пакетов не обновлено.  
Необходимо получить 0Б/12,8МБ архивов. После распаковки 37,4МБ будет занято.  
Хотите продолжить? [Y/n/?]
```

Рис. 2.4 Установка пакетов из репозитория

Обратите внимание, **aptitude** предлагает вам в квадратных скобках три возможных варианта ответа на поставленный вопрос:

[Y/n/?] Y означает *Yes*, то есть согласие, n – это *No*, то есть отказ, а ? – это просьба вывести справку. Вам нужно ввести символ, соответствующий вашему выбору и нажать Enter. Однако часто есть вариант по умолчанию, выделенный в списке большой буквой, и если вам нужен именно он, то вы можете ничего не вводить, просто нажать Enter.

Аналогично установке, удалить пакеты можно одной из двух команд:

```
sudo aptitude remove имя_пакета1 [имя_пакета2 ...]  
sudo aptitude purge имя_пакета1 [имя_пакета2 ...]
```

Первая удаляет только файлы пакета, оставляя пользовательские настройки нетронутыми, вторая же удаляет пакет полностью.

Посмотреть описание конкретного пакета можно командой

```
aptitude show имя_пакета
```

Произвести поиск нужного пакета по доступным источникам приложений можно командой

aptitude search фразы

По умолчанию поиск производится по именам пакетов, для поиска по описаниям надо перед искомой фразой добавить символ ~d:

aptitude search ~d фразы

aptitude имеет мощный графический интерфейс, попасть в него можно набрав в терминале просто aptitude. Вот как это выглядит:

Несмотря на неприглядный вид работать с этим очень удобно.

Обычно в разнообразных инструкциях для установки пакетов предлагается как раз использовать команду

sudo aptitude install имя_пакета

Это ни в коей мере не значит, что обязательно надо исполнять эту команду. Вы спокойно можете поставить указанные пакеты через тот же **Synaptic**. Просто авторы инструкций обычно экономят свое время на объяснении, куда и как надо нажимать в **Synaptic**, давая вместо всего этого одну маленькую команду. Но в конечном итоге **aptitude** и **Synaptic** выполняют одни и те же действия.

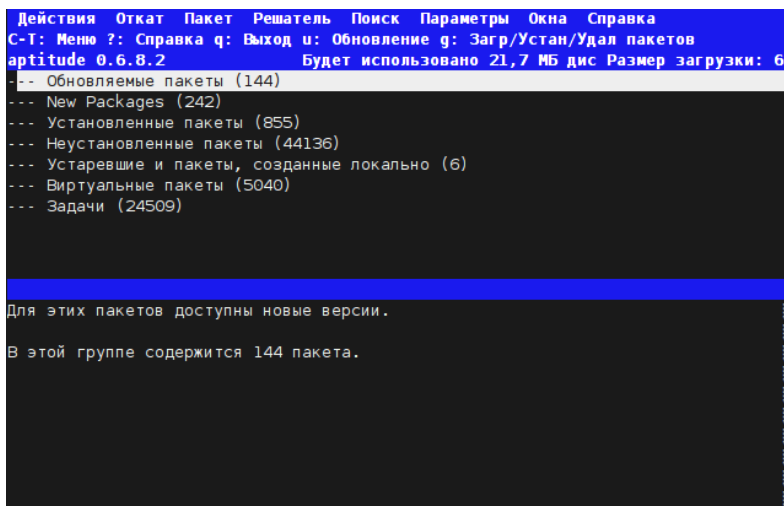


Рис. 2.5 Интерфейс утилиты aptitude

Утилита apt-get

В инструкциях гораздо чаще вместо **aptitude** используется **apt-get**. **apt-get** – это стандартная утилита управления пакетами, используется она ровно так же, как и **aptitude**, только у нее нет графического интерфейса и поиска. То есть во всех командах с **install**, **remove**, **purge** можно вместо **aptitude** писать **apt-get**.

Порядок выполнения лабораторной работы

1. Войдите как суперпользователь. Создайте нового пользователя: имя, группа, права, название домашнего каталога (задаются).
2. Установите программу:
 - через менеджер пакетов,
 - с использованием командной строки.
 - установка из исходных кодов.
3. Создать архив разными способами.
4. Создать сжатый архив.
5. Распаковать архив.

Контрольные вопросы

1. Дайте определение функциям администратора и суперпользователя.
2. Опишите процесс регистрации нового пользователя.
3. Опишите процесс установки программ.
4. Опишите выполнение архивирования и использование архиваторов.
5. Копирование файлов на стример.
6. Команда CPIO (Copy In/Out).
7. Архивация со сжатием.
8. Менеджер пакетов Synaptic.
9. Консольные инструменты управления пакетами.

Лабораторная работа 3

ПРОЦЕССЫ В ОПЕРАЦИОННОЙ СИСТЕМЕ UNIX

Цели и задачи

Знакомство с процессной организацией UNIX-подобных систем. Изучение информационных команд отслеживания информации о процессах. Изучение различных типов процессов. Изучение информации о первичном процессе `init` и уровнях загрузки системы. Создание программы на языке Си, реализующей порождение и замещение процессов с использованием системных вызовов UNIX, запуск команд UNIX из пользовательской программы.

3.1 Понятие процесса UNIX. Его контекст. Многозадачность

Процессы в Linux, как и файлы, являются аксиоматическими понятиями. Обычно процесс отождествляют с запущенной программой. Будем считать, что процесс - это рабочая единица системы, которая что-то выполняет. Многозадачность - это возможность одновременного сосуществования нескольких процессов в одной системе.

Linux - многозадачная операционная система. Это означает, что процессы в ней работают одновременно. Естественно, это условная формулировка. Ядро Linux постоянно переключает процессы, то есть время от времени дает каждому из них сколько-нибудь процессорного времени. Переключение происходит довольно быстро, поэтому нам кажется, что процессы работают одновременно.

Одни процессы могут порождать другие процессы, образуя древовидную структуру. Порождающие процессы называются родителями или родительскими процессами, а порожденные - потомками или дочерними процессами. На вершине этого «дерева» находится процесс `init`, который порождается автоматически ядром в процессе загрузки системы.

Получение информации о процессах в системе

Для получения информации о процессах в системе наиболее часто используются утилиты `ps` и `top`. В Linux вся информация о динамике выполнения системы отражается в каталоге `/proc`, утилиты

ps и top собирают данные о запущенных процессах на основании информации, находящейся в этом каталоге.

Контекст процесса складывается из пользовательского контекста и контекста ядра, как изображено на рисунке 3.1.

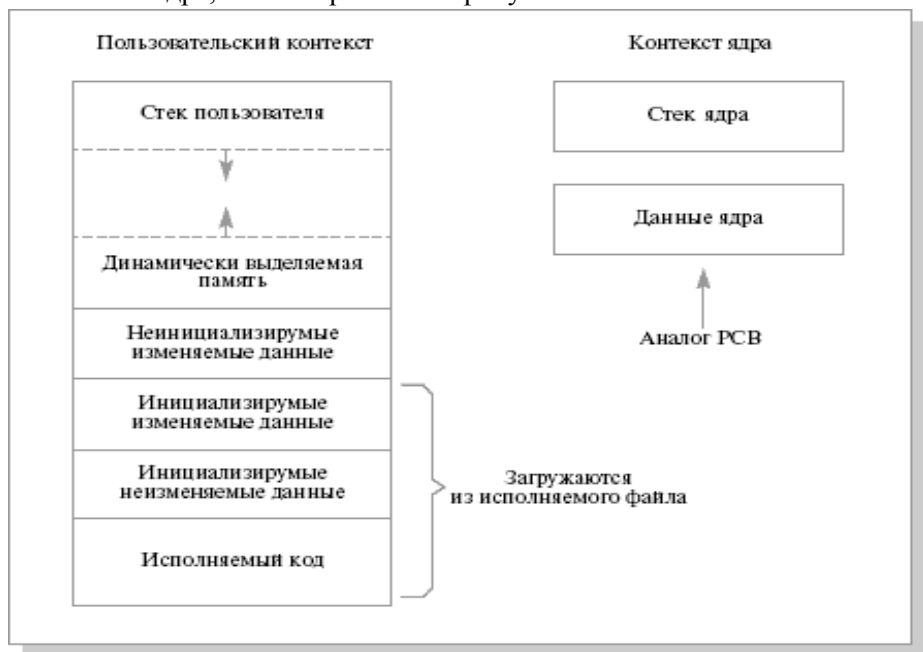


Рис. 3.1. Контекст процесса в UNIX

Под пользовательским контекстом процесса понимают код и данные, расположенные в адресном пространстве процесса. Все данные подразделяются:

- на инициализируемые неизменяемые данные (например, константы);
- инициализируемые изменяемые данные (все переменные, начальные значения которых присваиваются на этапе компиляции);
- не инициализируемые изменяемые данные (все статические переменные, которым не присвоены начальные значения на этапе компиляции);
- стек пользователя;
- данные, расположенные в динамически выделяемой памяти

(например, с помощью стандартных библиотечных C функций malloc(), calloc(), realloc()).

Исполняемый код и инициализируемые данные составляют содержимое файла программы, который выполняется в контексте процесса. Пользовательский стек применяется при работе процесса в пользовательском режиме (user-mode).

Под понятием «контекст ядра» объединяются системный контекст и регистровый контекст. Мы будем выделять в контексте ядра стек ядра, который используется при работе процесса в режиме ядра (kernel mode), и данные ядра, хранящиеся в структурах, являющихся аналогом блока управления процессом — PCB. Состав данных ядра будет уточняться на последующих семинарах. На этом занятии нам достаточно знать, что в данные ядра входят: идентификатор пользователя — UID, групповой идентификатор пользователя — GID, идентификатор процесса — PID, идентификатор родительского процесса — PPID.

3.2 Идентификация процесса. Иерархия процессов

Каждый процесс в операционной системе получает уникальный идентификационный номер – PID (process identifier). При создании нового процесса операционная система пытается присвоить ему свободный номер больший, чем у процесса, созданного перед ним. Если таких свободных номеров не оказывается (например, мы достигли максимально возможного номера для процесса), то операционная система выбирает минимальный номер из всех свободных номеров. В операционной системе Linux присвоение идентификационных номеров процессов начинается с номера 0, который получает процесс kernel при старте операционной системы. Этот номер впоследствии не может быть присвоен никакому другому процессу. Максимально возможное значение для номера процесса в Linux на базе 32-разрядных процессоров Intel составляет 231-1. Все процессы системы UNIX, кроме одного, создающегося при старте операционной системы, могут быть порождены только какими-либо другими процессами. Процессы с номерами 1 или 0 могут выступать в качестве прародителя всех остальных процессов в системах, подобных UNIX.

Таким образом, все процессы в UNIX связаны отношениями процесс-

родитель – процесс-потомок и образуют генеалогическое дерево процессов. Для сохранения целостности генеалогического дерева в ситуациях, когда процесс-родитель завершает свою работу до завершения выполнения процесса-потомка, идентификатор родительского процесса в данных ядра процесса-потомка (PPID – parent process identificator) изменяет свое значение на значение 1, соответствующее идентификатору процесса init, время жизни которого определяет время функционирования операционной системы. Тем самым процесс init как бы «усыновляет осиротевшие процессы».

3.3 Состояния процесса. Краткая диаграмма состояний

Модель состояний процессов в операционной системе UNIX представляет собой детализацию модели состояний. Краткая диаграмма состояний процессов в операционной системе UNIX изображена на рисунке 3.2.

Как мы видим, состояние процесса исполнение расщепилось на два состояния: исполнение в режиме ядра и исполнение в режиме пользователя. В состоянии «исполнение в режиме пользователя» процесс выполняет прикладные инструкции пользователя. В состоянии «исполнение в режиме ядра» выполняются инструкции ядра операционной системы в контексте текущего процесса (например, при обработке системного вызова или прерывания). Из состояния «исполнение в режиме пользователя» процесс не может непосредственно перейти в состояния «ожидание», «готовность» и «закончил исполнение». Такие переходы возможны только через промежуточное состояние «исполняется в режиме ядра». Также запрещен прямой переход из состояния «готовность» в состояние «исполнение в режиме пользователя».



Рис. 3.2 Сокращенная диаграмма состояний процесса в UNIX

3.4 Понятие системного вызова

В любой операционной системе поддерживается некоторый механизм, который позволяет пользовательским программам обращаться за услугами ядра ОС UNIX, такие средства называются системными вызовами. Смысл системных вызовов состоит в том, что для обращения к функциям ядра ОС используются «специальные команды» процессора, при выполнении которых возникает особого рода внутреннее прерывание процессора, переводящее его в режим ядра (в большинстве современных ОС этот вид прерываний называется *trap* - ловушка). При обработке таких прерываний ядро ОС распознает, что на самом деле прерывание является запросом к ядру со стороны пользовательской программы на выполнение определенных действий, выбирает параметры обращения и обрабатывает его, после чего выполняет «возврат из прерывания», возобновляя нормальное выполнение пользовательской программы. Понятно, что конкретные механизмы возбуждения внутренних прерываний по инициативе пользовательской программы различаются в разных аппаратных архитектурах. Поскольку ОС UNIX стремится обеспечить среду, в которой пользовательские программы могли бы быть полностью мобильны, потребовался дополнительный уровень, скрывающий особенности конкретного механизма возбуждения внутренних

прерываний. Этот механизм обеспечивается так называемой библиотекой системных вызовов.

Для пользователя библиотека системных вызовов представляет собой обычную библиотеку заранее реализованных функций системы программирования языка Си. При программировании на языке Си использование любой функции из библиотеки системных вызовов ничем не отличается от использования любой собственной или библиотечной Си-функции. Однако внутри любой функции конкретной библиотеки системных вызовов содержится код, являющийся, вообще говоря, специфичным для данной аппаратной платформы. Поведение всех программ в системе вытекает из поведения системных вызовов, которыми они пользуются. Сам термин «системный вызов» как раз означает «вызов системы для выполнения действия», т. е. вызов функции в ядре системы. Ядро работает в привилегированном режиме – режиме ядра, в котором имеет доступ к системным таблицам, регистрам и портам внешних устройств и диспетчера памяти, к которым обычным программам доступ аппаратно запрещен.

Системные вызовы `getuid` и `getgid`. Узнать идентификатор пользователя, запустившего программу на исполнение (UID), и идентификатор группы, к которой он относится (GID), можно с помощью системных вызовов `getuid()` и `getgid()`, применив их внутри этой программы.

Прототипы системных вызовов

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

```
#include <unistd.h>
```

```
uid_t getuid(void);
```

```
gid_t getgid(void);
```

Описание системных вызовов:

Системный вызов `getuid` возвращает идентификатор пользователя для текущего процесса.

Системный вызов `getgid` возвращает идентификатор группы пользователя для текущего процесса.

Типы данных `uid_t` и `gid_t` являются синонимами для одного из целочисленных типов языка Си.

Системные вызовы `getppid()` и `getpid()`. Данные ядра, находящиеся в контексте ядра процесса, не могут быть прочитаны процессом непосредственно. Для получения информации о них процесс должен совершить соответствующий системный вызов. Значение идентификатора текущего процесса может быть получено с помощью системного вызова `getpid()`, а значение идентификатора родительского процесса для текущего процесса – с помощью системного вызова `getppid()`. Системные вызовы не имеют параметров и возвращают идентификатор текущего процесса и идентификатор родительского процесса соответственно.

Прототипы системных вызовов

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

```
#include <unistd.h>
```

```
pid_t getpid(void);
```

```
pid_t getppid(void);
```

Описание системных вызовов

Системный вызов `getpid` возвращает идентификатор текущего процесса.

Системный вызов `getppid` возвращает идентификатор процесса-родителя для текущего процесса. Тип данных `pid_t` является синонимом для одного из целочисленных типов языка Си.

Пример 3-01 использования `getpid()` и `getppid()` для извлечения идентификаторов процессов.

```
#include <iostream>
```

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

```
#include <sys/wait.h>
```

```
#include <unistd.h>
```

```
#include <signal.h>
```

```
#include <cstdlib>
```

```
Using namespace std;
```

```
void mpinfo()
```

```
{
```

```

    cout<< "id process PID: " <<getpid() <<"\n";
    cout<< "id parent process PPID: " <<getppid()<<"\n";
    cout<< "id group process PGID: " <<getpgrp()<<"\n";
    cout<< "user real id UID: " <<getuid() <<"\n";
    cout<< "real id group user -GID:" <<getgid()<<"\n";
    cout<< "effect id user UID: " <<getuid()<<"\n";
    cout<< "effect id group GID: " <<getgid()<<"\n";
}
int main()
{
    int i,st;
    signal(SIGCHLD, SIG_IGN);
    for(i=1;i<=3;i++)
    {
        fork();
        mpinfo();
        cout<< "*****" <<"\n";//
        wait(&st);// ожидание завершения процесса
        exit(0);
    }
}

```

При запуске данной программы мы видим, что у каждого процесса есть несколько типов идентификаторов, основные типы – это реальные и эффективные. Реальный идентификатор пользователя (или группы) сообщает, кто создал процесс, а эффективный идентификатор пользователя (или группы) сообщает, от чьего лица выполняется процесс, если эта информация изменяется. На примере GID и effectid user обычно не меняются, потому что запускаете программу и выполняете ее только вы (1000). Также стоит обратить внимание на идентификаторы PPID (идентификатор процесса родителя) у процесса-потомка: это значение равно PID родителя. При этом существует еще один тип идентификаторов IDGROUPPROCESS, родители и потомки относятся к одной группе процессов.

3.5 Компиляция программ на языке Си в UNIX и запуск их на исполнение

Для компиляции программ в Linux применяются компиляторы gcc и g++. Для нормальной работы компилятора необходимо, чтобы исходные файлы, содержащие текст программы, имели имена, заканчивающиеся на .c.

В простейшем случае откомпилировать программу можно, запуская компилятор командой

g++ имя_исходного_файла.

Если программа была написана без ошибок, то компилятор создаст исполняемый файл с именем a.out.

Изменить имя создаваемого исполняемого файла можно, задав его с помощью опции -o:

g++ имя_исходного_файла -o имя_исполняемого_файла.

Компилятор g++ имеет несколько сотен возможных опций. Получить информацию о них возможно в UNIX Manual.

Запустить программу на исполнение можно, набрав имя исполняемого файла и нажав клавишу <Enter>.

3.6 Создание процесса в UNIX. Системный вызов fork()

Программное порождение процессов осуществляется с использованием системного вызова ***fork()*** – после выполнения этого вызова в системе появляется процесс, который является почти точной копией процесса, выдавшего данный системный вызов. У порожденного процесса по сравнению с родительским процессом (на уровне уже полученных знаний) изменяются значения следующих параметров:

- идентификатор процесса – PID;
- идентификатор родительского процесса – PPID.

Дополнительно может измениться поведение порожденного процесса по отношению к некоторым сигналам.

Системный вызов для порождения нового процесса

Прототип системного вызова

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

```
#include <unistd.h>
```

```
pid_t fork(void);
```

Описание системного вызова

Процесс, который инициировал системный вызов `fork`, принято называть родительским процессом (parent process). Вновь порожденный процесс принято называть процессом-ребенком (child process). Процесс-ребенок является почти полной копией родительского процесса. У порожденного процесса по сравнению с родительским изменяются значения следующих параметров:

1. идентификатор процесса;
 2. идентификатор родительского процесса;
 3. время, оставшееся до получения сигнала SIGALRM ;
 4. сигналы, ожидавшие доставки родительскому процессу, не будут доставляться порожденному процессу. Процесс-родитель и процесс-потомок разделяют один и тот же кодовый сегмент.
- Системный вызов `fork()` в случае успеха возвращает родительскому процессу идентификатор потомка, а потомку 0. В ситуации, когда процесс не может быть создан, функция `fork()` возвращает -1. В процессе выполнения системного вызова `fork()` порождается копия родительского процесса и возвращение из системного вызова будет происходить уже как в родительском, так и в порожденном процессах. Этот системный вызов является единственным, который вызывается один раз, а при успешной работе возвращается два раза (один раз в процессе-родителе и один раз в процессе-ребенке)! После выхода из системного вызова оба процесса продолжают выполнение регулярного пользовательского кода, следующего за системным вызовом.

Для иллюстрации сказанного рассмотрим следующие примеры программ.

Программа 3-02. Пример создания нового процесса с одинаковой работой процессов ребенка и родителя

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
    pid_t pid, ppid;
    int a = 0;
    (void) fork();
    /* При успешном создании нового процесса с этого места
    псевдопараллельно начинают работать два процесса: старый и
    новый */
    /* Перед выполнением следующего выражения значение переменной
    a в обоих процессах равно 0 */
    a = a + 1;
    /* Узнаем идентификаторы текущего и родительского процесса (в
    каждом из процессов!!!) */
    pid = getpid();
    ppid = getppid();
    /* Печатаем значения PID, PPID и вычисленное значение
    переменной a (в каждом из процессов!!!) */
    printf("My pid = %d, my ppid = %d, result = %d\n", (int)pid, (int)ppid,
    a);
    return 0;
}
```

Листинг 3-02. Программа создания процесса

Программа 3-03. Пример создания нового процесса с распечаткой идентификаторов

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
```

```

#include <cstdlib>
#include <iostream>
#include <wait.h>
using namespace std;
int main()
{
    pid_t pid;
    int i;
    switch(pid=fork()) {
        case -1:
            perror("fork");
            exit(1);
        case 0:
            printf(" CHILD: This is child\n");
            printf(" CHILD: PID -- %d\n", getpid());
            printf(" CHILD: PID my parent %d\n",getppid());
            printf(" CHILD: write code return:");
            scanf("%d",&i);
            printf(" CHILD: exit! \n");
            exit(i);
        default:
            printf("PARENT: This is parent\n");
            printf("PARENT: My Pid -- %d\n", getpid());
            printf("PARENT: PID my child %d\n",pid);
            printf("PARENT: I wait my child \n");
            wait(0);
            printf("PARENT: Code return my child%d\n",WEXITSTATUS(i));
            printf("PARENT: exit! \n");
    }
}

```

Листинг 3-03. Программа создания процесса

В данном случае процесс разделяется на родительский процесс и на процесс-потомок, после чего процесс-потомок спрашивает код своего завершения и завершается, все это время процесс-родитель ждет его завершения. После завершения процесса-потомка родитель получает код завершения потомка, выводит его и завершается сам.

3.7 Завершение процесса. Функция `exit()`

Существует два способа корректного завершения процесса в программах, написанных на языке Си. Первый способ мы использовали до сих пор: процесс корректно завершался по достижении конца функции `main()` или при выполнении оператора `return` в функции `main()`. Второй способ применяется при необходимости завершить процесс в каком-либо другом месте программы. Для этого используется функция `exit()` из стандартной библиотеки функций для языка С. При выполнении этой функции происходит сброс всех частично заполненных буферов ввода-вывода с закрытием соответствующих потоков, после чего инициируется системный вызов прекращения работы процесса и перевода его в состояние «закончил исполнение».

Возврата из функции в текущий процесс не происходит и функция ничего не возвращает.

Значение параметра функции `exit()` – кода завершения процесса – передается ядру операционной системы и может быть затем получено процессом, породившим завершившийся процесс. На самом деле при достижении конца функции `main()` также неявно вызывается эта функция со значением параметра 0.

Функция для нормального завершения процесса

Прототип функции

```
#include <stdlib.h>  
void exit(int status);
```

Параметры функции `main()` в языке Си. Переменные среды и аргументы командной строки

У функции `main()` в языке программирования Си существует три параметра, которые могут быть переданы ей операционной системой.

```
void main(int argc, char *argv[], char *envp[]);
```

Если вы наберете команду

\$ a.out a1 a2 a3, где **a.out** – имя запускаемой вами программы, то функция `main` программы из файла **a.out** вызовется с:


```
argc = 4 /* количество аргументов */  
argv[0] = "a.out" argv[1] = "a1"  
argv[2] = "a2" argv[3] = "a3"  
argv[4] = NULL
```

По соглашению *argv*[0] содержит имя выполняемого файла.

Первые два параметра при запуске программы на исполнение командной строкой позволяют узнать полное содержание командной строки. Вся командная строка рассматривается как набор слов, разделенных пробелами. Через параметр *argc* передается количество слов в командной строке, которой была запущена программа. Параметр *argv* является массивом указателей на отдельные слова. Так, например, если программа была запущена командой

```
$ ./a.out 12 abcd;
```

то значение параметра *argc* будет равно 3, *argv*[0] будет указывать на имя программы — первое слово — «a.out», *argv*[1] — на слово «12», *argv*[2] — на слово «abcd». Так как имя программы всегда присутствует на первом месте в командной строке, то *argc* всегда больше 0, а *argv*[0] всегда указывает на имя запущенной программы.

Анализируя в программе содержимое командной строки, мы можем предусмотреть ее различное поведение в зависимости от слов, следующих за именем программы. Таким образом, не внося изменений в текст программы, мы можем заставить ее работать по-разному от запуска к запуску. Например, компилятор *gcc*, вызванный командой *gcc* 1.c будет генерировать исполняемый файл с именем a.out, а при вызове командой *gcc* 1.c -o 1.exe – файл с именем 1.exe.

Третий параметр – *envp* – является массивом указателей на параметры окружающей среды процесса. Начальные параметры окружающей среды процесса задаются в специальных конфигурационных файлах для каждого пользователя и устанавливаются при входе пользователя в систему. В дальнейшем они могут быть изменены с помощью специальных команд операционной системы UNIX. Каждый параметр имеет вид: переменная=строка. Такие переменные используются для изменения долгосрочного поведения процессов, в отличие от аргументов командной строки. Например, задание параметра *TERM=vt100* может говорить процессам, осуществляющим вывод на экран дисплея, что работать им придется с терминалом vt100. Меняя значение переменной среды *TERM*, например на *TERM=console*, мы сообщаем таким процессам, что они должны изменить свое поведение

и осуществлять вывод для системной консоли.

Размер массива аргументов командной строки в функции `main()` мы получали в качестве ее параметра. Так как для массива ссылок на параметры окружающей среды такого параметра нет, то его размер определяется другим способом. Последний элемент этого массива содержит указатель **NULL**.

3.8 Изменение пользовательского контекста процесса. Семейство функций для системного вызова `exec()`

Для изменения пользовательского контекста процесса применяется системный вызов **`exec()`**, который пользователь не может вызвать непосредственно. Вызов **`exec()`** заменяет пользовательский контекст текущего процесса на содержимое некоторого исполняемого файла и устанавливает начальные значения регистров процессора (в том числе устанавливает программный счетчик на начало загружаемой программы). Этот вызов требует для своей работы задания имени исполняемого файла, аргументов командной строки и параметров окружающей среды. Для осуществления вызова программист может воспользоваться одной из шести функций: **`execlp()`**, **`execvp()`**, **`execl()`** и **`execv()`**, **`execle()`**, **`execve()`**, отличающихся друг от друга представлением параметров, необходимых для работы системного вызова **`exec()`**. Взаимосвязь указанных выше функций изображена на рисунке 3.3.



Рис. 3.3. Взаимосвязь различных функций для выполнения системного вызова `exec()`

Системный вызов `execve()` объявляется в заголовочном файле `unistd.h`:

`int execve (const char * path, char const * argv[], char * const envp[]);`

Все очень просто: системный вызов **`execve()`** заменяет текущий образ процесса программой из файла с именем **`path`**, набором аргументов **`argv`** и окружением **`envp`**. Здесь следует только учитывать, что **`path`** - это не просто имя программы, а путь к ней. Иными словами, чтобы запустить **`ls`**, нужно в первом аргументе указать «**`/bin/ls`**».

Массивы строк **`argv`** и **`envp`** обязательно должны заканчиваться элементом **`NULL`**. Кроме того следует помнить, что первый элемент массива **`argv`** (**`argv[0]`**) - это имя программы или что-либо иное. Непосредственные аргументы программы отсчитываются от элемента с номером 1.

В случае успешного завершения **`execve()`** ничего не возвращает, поскольку новая программа получает полное и безвозвратное управление текущим процессом. Если произошла ошибка, то по традиции возвращается -1.

Прототипы функций

`#include <unistd.h>`

***`int execl(const char *file,
const char *arg0,
... const char *argN,(char *)NULL)`***

`int execvp(const char *file, char *argv[])`

***`int execl(const char *path,
const char *arg0,
... const char *argN,(char *)NULL)`***

`int execv(const char *path, char *argv[])`

***`int execl(const char *path,
const char *arg0,
... const char *argN,(char *)NULL,
char * envp[])`***

`int execve(const char *path, char *argv[],`

*char *envp[]*)

Описание функций

Аргумент **file** является указателем на имя файла, который должен быть загружен. Аргумент **path** – это указатель на полный путь к файлу, который должен быть загружен.

Аргументы **arg0**, ..., **argN** представляют собой указатели на аргументы командной строки. Заметим, что аргумент **arg0** должен указывать на имя загружаемого файла. Аргумент **argv** представляет собой массив из указателей на аргументы командной строки. Начальный элемент массива должен указывать на имя загружаемой программы, а заканчиваться массив должен элементом, содержащим указатель **NULL**.

Аргумент **envp** является массивом указателей на параметры окружающей среды, заданные в виде строк «переменная=строка». Последний элемент этого массива должен содержать указатель **NULL**. Поскольку вызов функции не изменяет системный контекст текущего процесса, загруженная программа унаследует от загрузившего ее процесса следующие атрибуты:

1. идентификатор процесса;
2. идентификатор родительского процесса;
3. групповой идентификатор процесса;
4. идентификатор сеанса;
5. время, оставшееся до возникновения сигнала SIGALRM ;
6. текущую рабочую директорию;
7. маску создания файлов;
8. идентификатор пользователя;
9. групповой идентификатор пользователя;
10. явное игнорирование сигналов;
11. таблицу открытых файлов (если для файлового дескриптора не устанавливался признак «закрыть файл при выполнении exec()»).

Поскольку системный контекст процесса при вызове **exec()** остается практически неизменным, большинство атрибутов процесса, доступных пользователю через системные вызовы, после запуска новой программы также не изменяется.

Важно понимать разницу между системными вызовами **fork()**

и `exec()`. Системный вызов `fork()` создает новый процесс, у которого пользовательский контекст совпадает с пользовательским контекстом процесса-родителя. Системный вызов `exec()` изменяет пользовательский контекст текущего процесса, не создавая новый процесс.

Для иллюстрации использования системного вызова `exec()` рассмотрим следующие программы

Программа 3-04, изменяющая пользовательский контекст процесса (запускающая другую программу)

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char *argv[],
        char *envp[])
{
    /*Мы будем запускать команду cat с аргументом командной строки
    03-2.c без изменения параметров среды, т.е. фактически выполнять
    команду "cat 03-2.c", которая должна выдать содержимое данного
    файла на экран. Для функции execl в качестве имени программы мы
    указываем ее полное имя с путем от корневой директории —/bin/cat.
    Первое слово в командной строке у нас должно совпадать с именем
    запускаемой программы. Второе слово в командной строке – это имя
    файла, содержимое которого мы хотим распечатать. */

    (void) execl("/bin/cat", "/bin/cat",
        "03-2.c", NULL, envp);

    /* Сюда попадаем только при возникновении ошибки */
    printf("Error on program start\n");
    exit(-1);
    return 0;
}

Листинг 3-04. Программа, изменяющая пользовательский контекст процесса.
```

Программа 3-05, изменяющая пользовательский контекст процесса, пример использования функции `execv`

```
#include<iostream>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <signal.h>
#include <cstdlib>
using namespace std;
intmain()

{
    pid_tpid; // прототип системного вызова
    pid=fork();// тут происходит разделение
    switch (pid)
    {
        case -1:
            cout<<"error";// в случае возврата fork`ом -1, это считается ошибкой
            break;
        case 0:
            execlp("/bin/ps", "ps", NULL);// процесс-потомок
            //выводит запущенные процессы
            exit(0);
        default:
            wait(0);// процесс-родитель ждет завершения
            //процесса-потомка
            execlp("who", "who", NULL);//процессу-родителю
            //поручаем вывести пользователей
            exit(0);
    }
    return 0;
}
```

Листинг 3-05. Программа, изменяющая контекст процесса

Порядок выполнения работы

1. Рассмотреть утилиты `ps` и `top`. Для утилиты `ps` рассмотреть основные ключи, используя `man ps`, `ps --help`. Выполнить команду `ps` со всеми основными ключами. Вывести иерархию процессов с подробным описанием. Вывести все системные процессы. Рассмотреть в дереве процессов различные типы процессов. Посмотреть место процесса `init` в иерархии процессов, посмотреть уровень загрузки системы по умолчанию. В каталоге `/etc` найти файл `inittab` и рассмотреть его формат. Объяснить, что получится в результате выполнения утилиты `/sbin/init` с параметрами `1, 3, 5, 6, 0`;
2. Рассмотреть компиляцию программ с использованием компиляторов `gcc` и `g++`;
3. Изучить функции управления процессами (использовать `man`). Написать многопроцессную программу на Си или C++, реализующую запуск команд UNIX из процессов. Реализовать синхронизацию процессов-родителей с процессами-потомками (использовать функцию `wait`).
4. В качестве примера использования системных вызовов `getpid()` и `getppid()` самостоятельно напишите программу, печатающую значения `PID` и `PPID` для текущего процесса. Запустите ее несколько раз подряд. Посмотрите, как меняется идентификатор текущего процесса. Объясните наблюдаемые изменения.
5. Наберите программы, представленные в тексте лабораторного практикума, откомпилируйте и запустите на исполнение. Проанализируйте полученные результаты.
6. В качестве примера самостоятельно напишите программу, распечатающую значения аргументов командной строки и параметров окружающей среды для текущего процесса.
7. Для закрепления полученных знаний модифицируйте программу, созданную при выполнении задания раздела "Написание, компиляция и запуск программы с использованием вызова `fork()` с разным поведением процессов-ребенка и родителя" так, чтобы порожденный процесс запускал на исполнение новую (любую) программу.

Варианты индивидуальных заданий

1. Написать программу, реализующую меню из трех пунктов: 1-й пункт: ввести пользователя и вывести на экран все процессы, запущенные данным пользователем; 2-й пункт: показать всех пользователей, в настоящий момент, находящихся в системе; 3-й пункт: завершение.
2. Написать программу, реализующую меню из трех пунктов: 1-й пункт: вывести всех пользователей, в настоящее время работающих в системе; 2-й пункт: послать сообщение пользователю, имя пользователя, терминал и сообщение вводятся с клавиатуры; 3-й пункт: завершение.
3. Написать программу, реализующую меню из трех пунктов: 1-й пункт: показать все процессы пользователя, запустившего данный командный файл; 2-й пункт: послать сигнал завершения процессу текущего пользователя (ввести PID процесса); 3-й пункт: завершение.
4. Написать командный файл, подсчитывающий количество активных терминалов пользователя (имя пользователя вводится с клавиатуры).
5. Написать командный файл, посылающий сообщение всем активным пользователям (сообщение находится в файле).
6. Написать командный файл, посылающий сигнал завершения процессам текущего пользователя. Символьная маска имени процесса вводится с клавиатуры.
7. Написать командный файл, подсчитывающий количество определенных процессов пользователя (ввести имя пользователя и название процесса).
8. Реализовать Меню из двух пунктов: 1-й пункт: определить количество запущенных данным пользователем процессов bash (предусмотреть ввод имени пользователя); 2-й пункт: завершить все процессы bash данного пользователя.
9. Реализовать Меню из трех пунктов: 1-й пункт: поиск файла в каталоге, <Имя файла> и <Имя каталога> вводятся пользователем; 2-й пункт: копирование одного файла в другой каталог, <Имя файла> и <Имя каталога> вводятся; 3-й пункт: завершение командного файла.

10 Написать командный файл, который в цикле по нажатию клавиши выводит информацию о системе, активных пользователях в системе, а для введенного имени пользователя выводит список активных процессов данного пользователя.

11 Реализовать командный файл, который при старте выводит информацию о системе, информацию о пользователе, запустившем данный командный файл, далее в цикле выводит список активных пользователей в системе – запрашивает имя пользователя и выводит список всех процессов `bash`, запущенных данным пользователем.

12 Реализовать командный файл, позволяющий в цикле посылать всем активным пользователям сообщение – сообщение вводится с клавиатуры. Командный файл при старте выводит имя компьютера, имя запустившего командный файл пользователя, тип операционной системы, IP-адрес машины.

13 Реализовать командный файл, позволяющий в цикле посылать всем активным пользователям (исключая пользователя, запустившего данный командный файл) сообщение – сообщение вводится с клавиатуры. Командный файл при старте выводит имя компьютера, имя запустившего командный файл пользователя, тип операционной системы, список загруженных модулей.

14 Реализовать командный файл, который при старте выводит информацию о системе, информацию о пользователе, запустившем данный командный файл, далее в цикле выводит список активных пользователей в системе – запрашивает имя пользователя и выводит список всех терминалов, на которых зарегистрирован этот пользователь.

15 Реализовать командный файл, который выводит: дату, информацию о системе, текущий каталог, текущего пользователя, настройки домашнего каталога текущего пользователя, далее в цикле выводит список активных пользователей – запрашивает имя пользователя и выводит информацию об активности данного пользователя.

16 Реализовать командный файл, который выводит: дату в формате «день – месяц – год – время», информацию о системе в формате: имя компьютера : версия ОС : IP адрес : имя текущего пользователя : текущий каталог, выводит настройки домашнего каталога текущего пользователя и основные переменные окружения. Далее в

цикле выводит список активных пользователей – запрашивает имя пользователя и выводит информацию об активности введенного пользователя.

17 Реализовать программу, реализующую символьное меню (в цикле): 1-й пункт: вывод полной информации о файлах каталога (ввести имя каталога для отображения); 2-й пункт: изменить атрибуты файла: файл вводится с клавиатуры по запросу, атрибуты, которые требуется установить, тоже вводятся. После изменения атрибутов вывести на экран расширенный список файлов для проверки установленных атрибутов; 3-й пункт: выход. При старте программа выводит информацию об имени компьютера, IP-адреса и список всех пользователей, зарегистрированных в данный момент на компьютере.

18 Реализовать программу, реализующую символьное меню(в цикле) 1-й пункт: вывод полной информации о файлах каталога (ввести имя каталога для отображения); 2-й пункт: создать командный файл: файл вводится с клавиатуры по запросу, далее изменяются атрибут файла на исполнение, затем вводится с клавиатуры строка, которую будет исполнять командный файл. После изменения атрибутов вывести на экран расширенный список файлов для проверки установленных атрибутов и запустить созданный командный файл; 3-й пункт; выход. При старте командный файл выводит информацию об имени компьютера, IP-адреса и список всех пользователей зарегистрированных в данный момент на компьютере.

19 Написать командный файл реализующий символьное меню: 1-й пункт: работа с информационными командами (реализовать все основные информационные команды); 2-й пункт: Копирование файлов: в этом пункте выводится информация о содержимом текущего каталога, далее предлагается интерфейс копирования файла: ввод имени файла и ввод каталога для копирования. По выполнению пункта выводится содержимое каталога, куда был скопирован файл, и выводится содержимое скопированного файла; 3-й пункт: выход .

Контрольные вопросы

1. Назовите компоненты операционной системы.
2. Какие действия по управлению процессами выполняет ОС?
3. Что такое процесс? Его контекст. Понятие многозадачности.
4. Что такое стек процесса?
5. Идентификация процесса. Иерархия процессов.
6. Опишите возможные состояния процесса. Краткая диаграмма состояний.
7. Понятие системного вызова. Перечислите системные вызовы, рассматриваемые в данной лабораторной работе.
8. Опишите системные вызовы `getuid()` и `getgid()`.
9. Опишите системные вызовы `getppid()` и `getpid()`.
10. Создание процесса в UNIX. Системный вызов `fork()`.
11. Завершение процесса. Функция `exit()`.
12. Изменение пользовательского контекста процесса.
13. Семейство функций для системного вызова `exec()`.
14. Предположим, вам нужно разработать новую компьютерную архитектуру, которая вместо использования прерываний осуществляет аппаратное переключение процессов. Какие сведения необходимы центральному процессору? Опишите возможное устройство аппаратного переключения процессов.
15. На всех ныне существующих компьютерах хотя бы часть обработчиков прерываний написаны на ассемблере. Почему?
16. Когда в результате прерывания или системного вызова управление передается операционной системе, используется, как правило, область стека ядра, отделенная от стека прерываемого процесса. Почему?

Лабораторная работа 4

ОРГАНИЗАЦИЯ ВЗАИМОДЕЙСТВИЯ ПРОЦЕССОВ С ПОМОЩЬЮ КАНАЛОВ

Цели и задачи

Изучить переназначение операций ввода-вывода. Научиться использовать каналы для организации взаимодействия процессов и их синхронизации.

4.1 Понятие о потоке ввода-вывода

Канал связи (англ. **channel, data line**) — система технических средств и среда распространения сигналов для передачи сообщений (не только данных) от источника к получателю (и наоборот). Канал связи, понимаемый в узком смысле (тракт связи), представляет только физическую среду распространения сигналов, например, физическую линию связи

Каналы связи в зависимости от способа передачи сигналов классифицируют на несколько видов. Симплексный канал направляет сигналы только в одном направлении. Полудуплексный канал позволяет передать сигналы в двух направлениях, но поочередно. Такая передача экономически целесообразна также в любых типах каналов при взаимодействии партнеров типа запрос-ответ, когда перед ответом необходимо время для обработки запроса.

Когда процесс создает канал, ядро устанавливает два файловых дескриптора для пользования этим каналом. Первый дескриптор используется, чтобы открыть путь ввода в канал (запись), в то время как второй применяется для получения данных из канала (чтение).

Данные, идущие через канал, проходят через ядро. В операционной системе каналы представлены корректным **inode** - индексным дескриптором, который существует в пределах самого ядра, а не в какой-либо физической файловой системе.

Среди всех категорий средств коммуникации наиболее употребительными являются каналы связи, обеспечивающие достаточно безопасное и достаточно информативное взаимодействие процессов.

Существует две модели передачи данных по каналам связи — поток ввода-вывода и сообщения. Из них более простой является

потокковая модель, в которой операции передачи/приема информации вообще не интересуются содержимым того, что передается или принимается. Вся информация в канале связи рассматривается как непрерывный поток байт, не обладающий никакой внутренней структурой.

Понятие о `pipe`. Системный вызов `pipe()`

Наиболее простым способом для передачи информации с помощью *потокковой модели* между различными процессами или даже внутри одного процесса в операционной системе UNIX является ***pipe*** (канал, труба, конвейер).

Важное отличие `pipe`'а от файла заключается в том, что прочитанная информация немедленно удаляется из него и не может быть прочитана повторно.

Pipe можно представить себе в виде трубы ограниченной емкости, расположенной внутри адресного пространства операционной системы, доступ к входному и выходному отверстию которой осуществляется с помощью системных вызовов. В действительности ***pipe*** представляет собой область памяти, недоступную пользовательским процессам напрямую, зачастую организованную в виде кольцевого буфера (хотя существуют и другие виды организации). По буферу при операциях чтения и записи перемещаются два указателя, соответствующие входному и выходному потокам. При этом выходной *указатель* никогда не может обогнать *входной* и наоборот. Для создания нового экземпляра такого кольцевого буфера внутри операционной системы используется *системный вызов `pipe()`*.

Прототип системного вызова

```
#include <unistd.h>
```

```
int pipe(int *fd);
```

Описание системного вызова

Системный вызов **`pipe`** предназначен для создания **`pipe`** внутри операционной системы.

Параметр **`fd`** является указателем на массив из двух целых переменных. При нормальном завершении вызова в первый элемент

массива – **fd[0]** – будет занесен файловый дескриптор, соответствующий выходному потоку данных **pipe** и позволяющий выполнять только операцию чтения, а во второй элемент массива – **fd[1]** – будет занесен файловый дескриптор, соответствующий входному потоку данных и позволяющий выполнять только операцию записи.

Возвращаемые значения

Системный вызов возвращает значение **0** при нормальном завершении и значение **-1** при возникновении ошибок.

В процессе работы *системный вызов* организует выделение области памяти под *буфер* и указатели и заносит информацию, соответствующую входному и выходному потокам данных, в два элемента *таблицы открытых файлов*, связывая тем самым с каждым *pip*'ом два *файловых дескриптора*. Для одного из них разрешена только операция чтения из *pip*'а, а для другого – только операция записи в *pipe*. Для выполнения этих операций мы можем использовать те же самые системные вызовы *read()* и *write()*, что и при работе с файлами. Естественно, по окончании использования входного или/и выходного потока данных, нужно закрыть соответствующий *поток* с помощью системного вызова *close()* для освобождения системных ресурсов. Необходимо отметить, что, когда все процессы, использующие *pipe*, закрывают все ассоциированные с ним *файловые дескрипторы*, *операционная система* ликвидирует *pipe*. Таким образом, время существования *pip*'а в системе не может превышать время жизни процессов, работающих с ним.

Программа 4-01.c, иллюстрирующая работу с pip'ом в рамках одного процесса.

Пример создания pip'а, записи в него данных, чтения из него и освобождению выделенных ресурсов. Прогон программы для pipe в одном процессе

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <iostream>
```

```

#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
using namespace std;
int main()
{
    int fd[2];
    size_t size;
    char string[] = "Hello, world!";
    char resstring[14];
    /* Создание pipe */
    if(pipe(fd) < 0)
    {
        /* Если создать pipe не удалось, вывод сообщения об ошибке
        и прекращение работы */
        cout<<"Can't create pipe"<< endl;
    }
    /* Запись в pipe 14 байт из массива, т.е. всю строку "Hello, world!"
    вместе с признаком конца строки */
    size = write(fd[1], string, 14);
    if(size != 14)
    {
        /* Если записалось меньшее количество байт, вывод сообщения
        об ошибке */
        cout << "Can't write all string"<<endl;
    }
    /* Чтение из pip'a 14 байт в другой массив, т.е. всю записанную
    строку */
    size = read(fd[0], resstring, 14);
    if(size < 0)
    {
        /* Если прочитать не получилось, вывод сообщения об ошибке */
        cout << "Can't read string"<<endl;
    }
    /* Вывод прочитанной строки */
    cout<< resstring<< endl;
    /* Закрывание входного потока*/
    if(close(fd[0]) < 0)

```

```

{
    cout << "Can't close input stream" << endl;
}
/* Закрытие выходного потока */
if(close(fd[1]) < 0)
{
    cout << "Can't close output stream" << endl;
}
return 0;
}

```

Листинг 4-01. Программа, иллюстрирующая работу с `pipe` в рамках одного процесса

4.2 Организация связи через `pipe` между процессом-родителем и процессом-потомком. Наследование файловых дескрипторов при вызовах `fork()` и `exec()`

Таблица открытых файлов наследуется процессом-ребенком при порождении нового процесса системным вызовом `fork()` и входит в состав неизменяемой части системного контекста процесса при системном вызове `exec()` (за исключением тех потоков данных, для файловых дескрипторов которых был специальными средствами выставлен признак, побуждающий операционную систему закрыть их при выполнении `exec()`). Это обстоятельство позволяет организовать передачу информации через `pipe` между родственными процессами, имеющими общего прародителя, создавшего `pipe`.

Программа 4-02.с для организации однонаправленной связи между родственными процессами через `pipe`.

Рассмотрим программу, осуществляющую однонаправленную связь между процессом-родителем и процессом-ребенком: Процесс-родитель создает канал и порождает дочерний процесс, который посылает свой идентификатор (`pid`) в канал. Родительский процесс считывает данные из канала и выводит их на экран.


```

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <cstdlib>
#include <iostream>
using namespace std;
int main()
{
    char s[15] ; //для передачи и получения данных
    int fd[2] ; //для получения дескрипторов канала
    if (pipe(fd)<0) //проверка, открылся ли канал
    { fprintf (stdout,"error");
        return 0;
    } // если не удалось создать канал
    if (fork()==0)
    {
        int r = sprintf(s,"CHILD_Pid=%d", getpid()); // определение,
        какие данные записывать
        cout << "I am child, MyPid--"<< getpid()<<endl; // вывод на
        экран
        write(fd[1],&s,r); // запись данных в канал
        exit(0);
        return 1 ;
    }
    wait(0); //ожидание завершения процесса-потомка
    read(fd[0],&s,15); // считывание из канала данных процесса-потомка
    fprintf(stdout,"n I am Parent, I read your pid-  "%s\n", s);
    //вывод на экран того, что получилось
    close(fd[0]); //закрытие канала чтения
    close(fd[1]); //закрытие канала записи
    return 1 ;
}

```

Листинг 4-02. Программа работы с неименованным каналом

Программа 4-03.с, осуществляющая однонаправленную связь через pipe между процессом-родителем и процессом-ребенком

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <iostream>
using namespace std;
int main()
{
    int fd[2], result;
    size_t size;
    char resstring[14];
    /* Создание pipe */
    if(pipe(fd) < 0)
    {
        /* Если создать pipe не удалось, вывод об этом сообщения
        и прекращение работы */
        cout<<"Can't create pipe"<<endl;
    }
    /* Порождение нового процесса */
    result = fork();
    if(result)
    {
        /* Если создать процесс не удалось, вывод об этом сообщения
        и прекращение работы */
        cout<<" Can't fork child "<<endl;
    }
    else if (result > 0)
    {
        /* Закрытие выходного потока данных */
        close(fd[0]);
        /*Запись в pipe 14 байт, т.е. всю строку "Hello, world!" вместе с
        признаком конца строки */
        size = write(fd[1], "Hello, world!", 14);
        if(size != 14)
```

```

{
    /* Если записалось меньшее количество байт, вывод об этом
    сообщения и прекращение работы */
    cout<<" Can't write all string "<<endl
}
/* Закрытие входного потока данных, на этом родитель
прекращает работу */
close(fd[1]);
cout<<"Parent exit"<<endl;
}
else
{
    /* Порожденный процесс унаследовал от родителя таблицу
    открытых файлов и, зная файловые дескрипторы,
    соответствующие pip, может его использовать. Закрытие входного
    потока данных*/
    close(fd[1]);
    /* Чтение из pip'a 14 байт в массив, т.е. всю записанную строку */
    size = read(fd[0], resstring, 14);
    if(size < 0)
    {
        /* Если прочитать не удалось, вывод об этом сообщения и
        прекращение работы */
        cout<<" Can't read string "<<endl
    }
    /* Печать прочитанной строки */
    cout<< resstring<<endl;
    /* Закрытие входного потока и завершение работы */
    close(fd[0]);
}
return 0;
}

```

Листинг 4-03 Программа, осуществляющая однонаправленную связь через pipe между процессом-родителем и процессом-ребенком

4.3 Именованные каналы (FIFO — каналы). Понятие FIFO. Использование системного вызова `mknod()` для создания FIFO. Функция `mkfifo()`

Для организации потокового взаимодействия любых процессов в операционной системе UNIX применяется средство связи, получившее название **FIFO** (от **F**irst **I**nput **F**irst **O**utput) или именованный **pipe**. **FIFO** во всем подобен `pipe`'у, за одним исключением: данные о расположении **FIFO** в адресном пространстве ядра и его состоянии процессы можно получать не через родственные связи, а через файловую систему. Для этого при создании именованного `pipe`'а на диске заводится файл специального типа, обращаясь к которому, процессы могут получить интересующую их информацию. Для создания **FIFO** используется системный вызов `mknod()` или существующая в некоторых версиях UNIX функция `mkfifo()`.

Следует отметить, что при их работе не происходит действительного выделения области адресного пространства операционной системы под именованный **pipe**, а только заводится файл-метка, существование которой позволяет осуществить реальную организацию **FIFO** в памяти при его открытии с помощью уже известного нам системного вызова `open()`.

После открытия именованный `pipe` ведет себя точно так же, как и неименованный. Для дальнейшей работы с ним применяются системные вызовы `read()`, `write()` и `close()`. Время существования **FIFO** в адресном пространстве ядра операционной системы, как и в случае с `pipe`'ом, не может превышать время жизни последнего из использовавших его процессов. Когда все процессы, работающие с **FIFO**, закрывают все файловые дескрипторы, ассоциированные с ним, система освобождает ресурсы, выделенные под **FIFO**. Вся непрочитанная информация теряется. В то же время файл-метка остается на диске и может использоваться для новой реальной организации **FIFO** в дальнейшем.

Использование системного вызова **mknod** для создания **FIFO**

Прототип системного вызова

#include <sys/stat.h>

#include <unistd.h>

int mknod(char *path, int mode, int dev);

Описание системного вызова

Параметр **dev** является несущественным в нашей ситуации, и мы будем всегда задавать его равным 0.

Параметр **path** является указателем на строку, содержащую полное или относительное имя файла, который будет являться меткой **FIFO** на диске. Для успешного создания **FIFO** файла с таким именем перед вызовом существовать не должно.

Параметр **mode** устанавливает атрибуты прав доступа различных категорий пользователей к **FIFO**. Этот параметр задается как результат побитовой операции "или" значения **S_IFIFO**, указывающего, что системный вызов должен создать **FIFO**, и некоторой суммы следующих восьмеричных значений:

0400 – разрешено чтение для пользователя, создавшего **FIFO**;

0200 – разрешена запись для пользователя, создавшего **FIFO**;

0040 – разрешено чтение для группы пользователя, создавшего **FIFO**;

0020 – разрешена запись для группы пользователя, создавшего **FIFO**;

0004 – разрешено чтение для всех остальных пользователей;

0002 – разрешена запись для всех остальных пользователей.

При создании **FIFO** реально устанавливаемые права доступа получаются из стандартной комбинации параметра **mode** и маски создания файлов текущего процесса **umask**, а именно – они равны **(0777 & mode) & ~umask**.

Возвращаемые значения

При успешном создании **FIFO** системный вызов возвращает значение 0, при неуспешном – отрицательное значение.

Функция **mkfifo**

Прототип функции

#include <sys/stat.h>

#include <unistd.h>

int **mkfifo**(char *path, int mode);

Описание функции

Функция **mkfifo** предназначена для создания FIFO в операционной системе.

Параметр **path** является указателем на строку, содержащую полное или относительное имя файла, который будет являться меткой FIFO на диске. Для успешного создания FIFO файла с таким именем перед вызовом функции не должно существовать.

Параметр **mode** устанавливает атрибуты прав доступа различных категорий пользователей к FIFO. Этот параметр задается как некоторая сумма следующих восьмеричных значений:

0400 – разрешено чтение для пользователя, создавшего FIFO;

0200 – разрешена запись для пользователя, создавшего FIFO;

0040 – разрешено чтение для группы пользователя, создавшего FIFO;

0020 – разрешена запись для группы пользователя, создавшего FIFO;

0004 – разрешено чтение для всех остальных пользователей;

0002 – разрешена запись для всех остальных пользователей.

При создании FIFO реально устанавливаемые права доступа получаются из стандартной комбинации параметра **mode** и маски создания файлов текущего процесса **umask**, а именно – они равны $(0777 \& \text{mode}) \& \sim \text{umask}$.

Возвращаемые значения

При успешном создании FIFO функция возвращает значение 0, при неуспешном – отрицательное значение.

Важно понимать, что файл типа FIFO не служит для размещения на диске информации, которая записывается в именованный **pipe**. Эта информация располагается внутри адресного пространства операционной системы, а файл является только меткой, создающей предпосылки для ее размещения.

Не пытайтесь просмотреть содержимое этого файла с помощью Midnight Commander (mc)!!! Это приведет к его глубокому зависанию!

Для иллюстрации взаимодействия процессов через FIFO рассмотрим такую программу:

Программа 4-04.с. FIFO в родственных процессах.

Программа, осуществляющая однонаправленную связь через FIFO между процессом-родителем и процессом-ребенком

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <iostream>
using namespace std;
int main()
{
    int fd, result;
    size_t size;
    char resstring[14];
    char name[]="aaa.fifo";
    /* Обнуление маски создания файлов текущего процесса для того,
    чтобы права доступа у создаваемого FIFO точно соответствовали
    параметру вызова mknod() */
    (void)umask(0);
    /* Создание FIFO с именем aaa.fifo в текущей директории */
    if(mknod(name, S_IFIFO | 0666, 0) < 0)
    {
        /* Если создать FIFO не удалось, вывод об этом сообщения и
        прекращение работы */
        cout<<"Can't create FIFO"<<endl;
    }
    /* Порождение нового процесса */
    if((result = fork()) < 0)
    {
        /* Если создать процесс не удалось, вывод об этом сообщения и
```

```

прекращение работы */
    cout<<"Can't fork child"<<endl;
}
else if (result > 0)
{
    /* В родительском процессе открываем FIFO и передаем
    информацию процессу-ребенку*/
    if((fd = open(name, O_WRONLY)) < 0)
    {
        /* Если открыть FIFO не удалось, вывод об этом сообщения и
        прекращение работы */
        cout<<"Can't open FIFO for writing"<<endl;
    }
    /* Записать в FIFO 14 байт, т.е. всю строку "Hello, world!"
    вместе с признаком конца строки */
    size = write(fd, "Hello, world!", 14);
    if(size != 14)
    {
        /* Если записалось меньшее количество байт, вывод об этом
        сообщения
        и прекращение работы */
        cout<<"Can't write all string to FIFO"<<endl;
    }
    /* Закрытие входного потока данных и на этом родитель
    прекращает работу */
    close(fd);
    cout<<"Parent exit"<<endl;
}
else
{
    /* Открытие FIFO на чтение.*/
    if((fd = open(name, O_RDONLY)) < 0)
    {
        /* Если открыть FIFO не удалось, печатаем об этом
        сообщение и прекращаем работу */
        cout<<"Can't open FIFO for reading "<<endl;
    }
    /* Чтение из FIFO 14 байт в массив, т.е. всю записанную строку

```



```

*/
size = read(fd, resstring, 14);
if(size < 0)
{
    /* Если прочитать не смогли, вывод об этом сообщения
    и прекращение работы */
    cout<<"Can't read string"<<endl;
}
/* Печать прочитанной строки */
cout<<resstring<< endl;
/* Закрытие входного потока и завершение работы */
close(fd);
}
return 0;
}

```

Листинг 4-04. Программа, осуществляющая однонаправленную связь через FIFO между процессом-родителем и процессом-ребенком

В этой программе информацией между собой обмениваются процесс-родитель и процесс-ребенок. Повторный запуск этой программы приведет к ошибке при попытке создания FIFO, так как файл с заданным именем уже существует. Здесь нужно либо удалять его перед каждым прогоном программы с диска вручную, либо после первого запуска модифицировать исходный текст, исключив из него все, связанное с системным вызовом `tknmod()`.

4.4 Открытие файла. Системный вызов `open()`

Файловый дескриптор используется в качестве параметра, описывающего поток ввода-вывода, для системных вызовов, выполняющих операции над этим потоком. Поэтому прежде чем совершать операции чтения данных из файла и записи их в файл, необходимо поместить информацию о файле в таблицу открытых файлов и определить соответствующий файловый дескриптор. Для этого применяется процедура открытия файла, осуществляемая системным вызовом `open()`.

Системный вызов `open`

Прототип системного вызова

`#include <fcntl.h>`

`int open(char *path, int flags);`

`int open(char *path, int flags, int mode);`

Описание системного вызова

Системный вызов `open` предназначен для выполнения операции открытия файла и, в случае ее удачного осуществления, возвращает файловый дескриптор открытого файла (небольшое неотрицательное целое число, которое используется в дальнейшем для других операций с этим файлом).

Параметр `path` является указателем на строку, содержащую полное или относительное имя файла.

Параметр `flags` может принимать одно из следующих трех значений:

O_RDONLY – если над файлом в дальнейшем будут совершаться только операции чтения;

O_WRONLY – если над файлом в дальнейшем будут осуществляться только операции записи;

O_RDWR – если над файлом будут осуществляться и операции чтения, и операции записи.

Каждое из этих значений может быть скомбинировано посредством операции «побитовое или (|)» с одним или несколькими флагами:

O_CREAT – если файла с указанным именем не существует, он должен быть создан;

O_EXCL – применяется совместно с флагом **O_CREAT**. При совместном их использовании и существовании файла с указанным именем, открытие файла не производится и констатируется ошибочная ситуация;

O_NDELAY – запрещает перевод процесса в состояние «ожидание» при выполнении операции открытия и любых последующих операциях над этим файлом;

O_APPEND – при открытии файла и перед выполнением каждой операции записи (если она, конечно, разрешена) указатель текущей позиции в файле устанавливается на конец файла;

O_TRUNC – если файл существует, уменьшить его размер до 0, с сохранением существующих атрибутов файла, кроме, быть может, времен последнего доступа к файлу и его последней модификации.

Кроме того, в некоторых версиях операционной системы UNIX могут применяться дополнительные значения флагов:

O_SYNC – любая операция записи в файл будет блокироваться (т. е. процесс будет переведен в состояние «ожидание») до тех пор, пока записанная информация не будет физически помещена на соответствующий нижележащий уровень hardware;

O_NOCTTY – если имя файла относится к терминальному устройству, оно не становится управляющим терминалом процесса, даже если до этого процесс не имел управляющего терминала.

Параметр **mode** устанавливает атрибуты прав доступа различных категорий пользователей к новому файлу при его создании. Он обязателен, если среди заданных флагов присутствует флаг **O_CREAT**, и может быть опущен в противном случае.

Этот параметр задается как сумма следующих восьмеричных значений:

0400 – разрешено чтение для пользователя, создавшего файл;

0200 – разрешена запись для пользователя, создавшего файл;

0100 – разрешено исполнение для пользователя, создавшего файл;

0040 – разрешено чтение для группы пользователя, создавшего файл;

0020 – разрешена запись для группы пользователя, создавшего файл;

0010 – разрешено исполнение для группы пользователя, создавшего файл;

0004 – разрешено чтение для всех остальных пользователей;

0002 – разрешена запись для всех остальных пользователей;

0001 – разрешено исполнение для всех остальных пользователей.

При создании файла реально устанавливаемые права доступа получаются из стандартной комбинации параметра **mode** и маски создания файлов текущего процесса **umask**, а именно – они равны **mode & ~umask**.

При открытии файлов типа **FIFO** системный вызов имеет некоторые особенности поведения по сравнению с открытием файлов других типов. Если **FIFO** открывается только для чтения и не задан флаг **O_NDELAY**, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет **FIFO** на запись. Если флаг **O_NDELAY** задан, то возвращается значение файлового дескриптора, ассоциированного с **FIFO**. Если **FIFO** открывается только для записи и не задан флаг **O_NDELAY**, то

процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет **FIFO** на чтение. Если флаг **O_NDELAY** задан, то констатируется возникновение ошибки и возвращается значение -1.

Возвращаемое значение

Системный вызов возвращает значение файлового дескриптора для открытого файла при нормальном завершении и значение -1 при возникновении ошибки.

4.5 Системные вызовы **read()**, **write()**

Для совершения потоковых операций чтения информации из файла и ее записи в файл применяются системные вызовы **read()** и **write()**.

Прототипы системных вызовов

```
#include <sys/types.h>
#include <unistd.h>
size_t read(int fd, void *addr,
             size_t nbytes);
size_t write(int fd, void *addr,
             size_t nbytes);
```

Описание системных вызовов

Системные вызовы **read** и **write** предназначены для осуществления потоковых операций ввода (чтения) и вывода (записи) информации над каналами связи, описываемыми файловыми дескрипторами, т.е. для файлов, **pipe**, **FIFO** и **socket**.

Параметр **fd** является файловым дескриптором созданного ранее потокового канала связи, через который будет отсылаться или получаться информация, т. е. значением, которое вернул один из системных вызовов **open()** , **pipe()** или **socket()**.

Параметр **addr** представляет собой адрес области памяти, начиная с которого будет браться информация для передачи или размещаться принятая информация.

Параметр **nbytes** для системного вызова **write** определяет количество байт, которое должно быть передано, начиная с адреса памяти **addr**.

Параметр **nbytes** для системного вызова **read** определяет количество байт, которое мы хотим получить из канала связи и разместить в памяти, начиная с адреса **addr**.

Возвращаемые значения

В случае успешного завершения системный вызов возвращает количество реально посланных или принятых байт. Это значение (больше или равное 0) может не совпадать с заданным значением параметра **nbytes**, а быть меньше, чем оно, в силу отсутствия места на диске или в линии связи при передаче данных или отсутствия информации при ее приеме. При возникновении какой-либо ошибки возвращается отрицательное значение.

Особенности поведения при работе с файлами

При работе с файлами информация записывается в файл или читается из файла, начиная с места, определяемого указателем текущей позиции в файле. Значение указателя увеличивается на количество реально прочитанных или записанных байт. При чтении информации из файла она не пропадает из него. Если системный вызов **read** возвращает значение 0, то это означает, что файл прочитан до конца.

4.6 Системный вызов **close()**

Прототип системного вызова

#include <unistd.h>

int close(int fd);

Описание системного вызова

Системный вызов **close** предназначен для корректного завершения работы с файлами и другими объектами ввода-вывода, которые описываются в операционной системе через файловые дескрипторы: **pipe**, **FIFO**, **socket**.

Параметр **fd** является дескриптором соответствующего объекта, т. е. значением, которое вернул один из системных вызовов **open()**, **pipe()** или **socket()**.

Возвращаемые значения

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

4.7 Особенности поведения вызова `open()` при открытии **FIFO**

Системные вызовы `read()` и `write()` при работе с **FIFO** имеют те же особенности поведения, что и при работе с **pip**'ом. Системный вызов `open()` при открытии **FIFO** ведет себя несколько иначе, чем при открытии других типов файлов, что связано с возможностью блокирования выполняющих его процессов. Если **FIFO** открывается только для чтения и флаг `O_NDELAY` не задан, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет **FIFO** на запись. Если флаг `O_NDELAY` задан, то возвращается значение файлового дескриптора, ассоциированного с **FIFO**. Если **FIFO** открывается только для записи и флаг `O_NDELAY` не задан, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет **FIFO** на чтение. Если флаг `O_NDELAY` задан, то констатируется возникновение ошибки и возвращается значение **-1**. Задание флага `O_NDELAY` в параметрах системного вызова `open()` приводит и к тому, что процессу, открывшему **FIFO**, запрещается блокировка при выполнении последующих операций чтения из этого потока данных и записи в него.

Для иллюстрации сказанного давайте рассмотрим следующую программу:

Программа 4-05.c для записи информации в файл.

Программа, иллюстрирующая использование системных вызовов `open()`, `write()` и `close()` для записи информации в файл

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <iostream>
using namespace std;
int main()
{
    int fd;
    size_t size;
```

```

char string[] = "Hello, world!";
/* Обнуление маски создания файлов текущего процесса для того,
чтобы права доступа у создаваемого файла точно соответствовали
параметру вызова open() */
(void)umask(0);
/* Открытие файла с именем myfile в текущей директории только
для операций вывода. Если файла не существует, то его создание с
правами доступа 0666, т. е. read-write для всех категорий
пользователей */
if((fd = open("myfile", O_WRONLY | O_CREAT, 0666)) < 0)
{
    /* Если файл открыть не удалось, вывод об этом сообщения и
прекращение работы */
    cout<<"Can't open file"<<endl;
}
/* Запись в файл 14 байт из нашего массива, т.е. всю строку "Hello,
world!" вместе с признаком конца строки */
size = write(fd, string, 14);
if(size == write(fd, string, 14))
{
    cout<<"File written"<<endl;
}
if(size != 14)
{
    /* Если записалось меньшее количество байт, то вывод
сообщения об ошибке */
    cout<<"Can't write all string"<<endl;
}
/* Закрывание файла */
if(close(fd) < 0)
{
    cout<<"Can't close file"<<endl;
}
return 0;
}

```

Листинг 4-05. Программа, иллюстрирующая использование системных вызовов `open()`, `write()` и `close()` для записи информации в файл

4.8 Особенности поведения вызовов `read()` и `write()` для `pip'a`

Системные вызовы `read()` и `write()` имеют определенные особенности поведения при работе с `pip'ом`, связанные с его ограниченным размером, задержками в передаче данных и возможностью блокирования обменивающихся информацией процессов.

При написании программ, обменивающихся большими объемами информации через `pipe`, необходимо проверять значения, возвращаемые вызовами. За один раз из `pip'a` может прочитаться меньше информации, чем вы запрашивали, и за один раз в `pipe` может записаться меньше информации, чем вам хотелось бы.

Одна из особенностей поведения блокирующегося системного вызова `read()` связана с попыткой чтения из пустого `pip'a`. Если есть процессы, у которых этот `pipe` открыт для записи, то системный вызов блокируется и ждет появления информации. Если таких процессов нет, он вернет значение 0 без блокировки процесса. Эта особенность приводит к необходимости закрытия файлового дескриптора, ассоциированного с входным концом `pip'a`, в процессе, который будет использовать `pipe` для чтения (`close (fd[1])` в процессе-ребенке в программе из раздела «Организация связи через `pipe` между процессом-родителем и процессом потомком»). Аналогичной особенностью поведения при отсутствии процессов, у которых `pipe` открыт для чтения, обладает и системный вызов `write()`, с чем связана необходимость закрытия файлового дескриптора, ассоциированного с выходным концом `pip'a`, в процессе, который будет использовать `pipe` для записи (`close (fd[0])` в процессе-родителе в той же программе).

Программа 4-06.с. Пример работы с именованным каналом между процессами одного приложения.

Процесс-родитель создает именованный канал и порождает дочерний процесс, который посылает свой идентификатор (`pid`) в канал. Родительский процесс считывает данные из канала и выводит их на экран.

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



```

#include <fcntl.h>
#include <sys/wait.h>
#include <iostream>

using namespace std;

int main()
{
    char s[15] ; //для передачи и получения данных
    int fd[2] ; //для получения дескрипторов канала
    if (mkfifo("mypipe",S_IFIFO|0666)<0) //если не удалось создать канал
    {
        cout << "\Can't create channel";
        return 0;
    }
    //получить дескриптор для чтения
    //если не указать флаг O_NONBLOCK, процесс заблокирует сам себя
    fd[0] = open("mypipe",O_RDONLY|O_NONBLOCK);

    //получить дескриптор для записи
    fd[1] = open("mypipe",O_WRONLY);
    if (fork()==0)
    {
        //программный код для дочернего процесса
        int r = sprintf(s,"MyPid=%d",getpid()) ; //записать свой pid в s, r—длина строки
        write(fd[1],&s,r); //послать данные s в канал через дескриптор для записи
        return 1;
    }
    wait(NULL); /*ожидание дочернего процесса необходимо, так как функция чтения из канала стала не блокирующей, т.е. если дочерний процесс не успеет записать данные в канал, функция чтения не получит данных и завершится*/
    read(fd[0],&s,15); //прочитать данные из канала через дескриптор для чтения

    cout << "Parent read"<< s<<endl; //вывести полученную строку на экран
}

```

```

close(fd[0]); //закрывать дескрипторы канала
close(fd[1]); //закрывать дескрипторы канала
unlink("mypipe"); //удалить канал
return 1;
}

```

Листинг 4-06. Программа работы с именованным каналом

Программа 4-07.с. Пример работы с именованным каналом между двумя приложениями: читатель и писатель.

Первое приложение создает именованный канал, второе приложение (писатель) получает дескриптор канала на запись и записывает в него свой идентификатор (pid). Первое приложение (читатель) ожидает поступления данных в канал и выводит их на экран. Для совместной работы приложений, исходя из особенностей алгоритма, необходимо сначала запустить первое приложение.

Читатель:

```

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <cstdlib>
#include <iostream>
#include <fcntl.h>
#include <sys/stat.h>
int main()
{
    char s[15];
    int fd;
    mkfifo("mypipe", S_IFIFO|0666);
    fd=open("mypipe", O_RDONLY);
    read(fd, &s, 15);
    fprintf(stdout, "READ: '%s'\n", s);
    close(fd);
}

```

```

unlink("mypipe");
return 1 ;
}

```

Писатель:

```

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <cstdlib>
#include <iostream>
#include <fcntl.h>
#include <sys/stat.h>
int main()
{
char s[15];
int fd;
fd=open("mypipe",O_WRONLY);
sprintf(s,"MyPid=%d",getpid());
write(fd,&s,15);
close(fd);
return 1;
}

```

Листинг 4-07 читатель и писатель

Работа осуществляется с двух консолей (программы заранее откомпилированы, изменения с методичкой минимальны), первая программа читатель, запускаю ее с 1 терминала, он ждет отклика от писателя. Со второго терминала мы запускаем писателя; после того как писатель передал информацию в канал, писатель закрывается, далее читатель получает информацию из канала выводит на экран и тоже закрывается.

Таблица 4.1 Особенности поведения при работе с pipe, FIFO и socket

Особенности поведения при работе с pipe, FIFO и socket	
Системный вызов read	
Ситуация	Поведение
Попытка прочитать меньше байт, чем есть в наличии в канале связи.	Читает требуемое количество байт и возвращает значение, соответствующее прочитанному количеству. Прочитанная информация удаляется из канала связи.
В канале связи находится меньше байт, чем затребовано, но не нулевое количество.	Читает все, что есть в канале связи, и возвращает значение, соответствующее прочитанному количеству. Прочитанная информация удаляется из канала связи.
Попытка читать из канала связи, в котором нет информации. Блокировка вызова разрешена.	Вызов блокируется до тех пор, пока не появится информация в канале связи и пока существует процесс, который может передать в него информацию. Если информация появилась, то процесс разблокируется, и поведение вызова определяется двумя предыдущими строками таблицы. Если в канал некому передать данные (нет ни одного процесса, у которого этот канал связи открыт для записи), то вызов возвращает значение 0. Если канал связи полностью закрывается для записи во время блокировки читающего процесса, то процесс разблокируется, и системный вызов возвращает значение 0.

Продолжение таблицы 4.1

Попытка читать из канала связи, в котором нет информации. Блокировка вызова не разрешена.	Если есть процессы, у которых канал связи открыт для записи, системный вызов возвращает значение -1 и устанавливает переменную errno в значение EAGAIN. Если таких процессов нет, системный вызов возвращает значение 0.
Системный вызов write	
Ситуация	Поведение
Попытка записать в канал связи меньше байт, чем осталось до его заполнения.	Требуемое количество байт помещается в канал связи, возвращается записанное количество байт.
Попытка записать в канал связи больше байт, чем осталось до его заполнения. Блокировка вызова разрешена.	Вызов блокируется до тех пор, пока все данные не будут помещены в канал связи. Если размер буфера канала связи меньше, чем передаваемое количество информации, то вызов будет ждать, пока часть информации не будет считана из канала связи. Возвращается записанное количество байт.
Попытка записать в канал связи больше байт, чем осталось до его заполнения, но меньше, чем размер буфера канала связи. Блокировка вызова запрещена.	Системный вызов возвращает значение -1 и устанавливает переменную errno в значение EAGAIN.
В канале связи есть место. Попытка записать в канал связи больше байт, чем осталось до его заполнения, и больше, чем размер буфера канала связи. Блокировка вызова запрещена.	Записывается столько байт, сколько осталось до заполнения канала. Системный вызов возвращает количество записанных байт.

Попытка записи в канал связи, в котором нет места. Блокировка вызова не разрешена.	Системный вызов возвращает значение -1 и устанавливает переменную errno в значение EAGAIN.
Попытка записи в канал связи, из которого некому больше читать, или полное закрытие канала на чтение во время блокировки системного вызова.	Если вызов был заблокирован, то он разблокируется. Процесс получает сигнал SIGPIPE. Если этот сигнал обрабатывается пользователем, то системный вызов вернет значение -1 и установит переменную errno в значение EPIPE.
Необходимо отметить дополнительную особенность системного вызова write при работе с рір'ами и FIFO. Запись информации, размер которой не превышает размер буфера, должна осуществляться атомарно – одним подряд лежащим куском. Этим объясняется ряд блокировок и ошибок в предыдущем перечне.	

Порядок выполнения лабораторной работы

1. Рассмотреть функции работы с неименованными каналами. Откомпилировать приведенные программы, привести скрины выполнения программ.
2. Рассмотреть функции работы с именованными каналами. Откомпилировать приведенные программы, привести скрины выполнения программ. Обратите внимание на использование системного вызова umask() с параметром 0 для того, чтобы права доступа к созданному файлу точно соответствовали указанным в системном вызове open() .
3. Системные вызовы open (), read(), write(), close(). Откомпилировать программу, иллюстрирующую использование системных вызовов open(), write() и close() для записи информации в файл. Привести скрин ее работы.
4. Системный вызов рір. Откомпилировать программы, иллюстрирующие работу вызова рір, скрины их реализации представить в отчете.
5. Пример работы с именованным каналом, откомпилируйте

- программу, скрины их реализации представьте в отчете
6. Для закрепления полученных знаний написать на базе Листинга 5 две программы, одна из которых пишет информацию в FIFO, а вторая – читает из него, так, чтобы между ними не было ярко выраженных родственных связей (т. е. чтобы ни одна из них не была потомком другой).
 7. Изменить программу 6 так, чтобы она читала записанную ранее в файл информацию и печатала ее на экране. Все лишние операторы желательно удалить.
 8. Разработать алгоритм решения задания а с учетом разделения вычислений между несколькими процессами. Для обмена информацией между процессами использовать неименованные каналы.
 9. Реализовать алгоритм решения задания а и протестировать на нескольких примерах.
 10. Разработать алгоритм решения задания б, разделив вычисления между несколькими приложениями. Для обмена информацией между приложениями использовать именованные каналы.
 11. Реализовать алгоритм решения задания б и протестировать на нескольких примерах.
 12. Написать приложение, переназначающее стандартный вывод в канал и стандартный ввод из канала и замещаемое приложением задания а.

Варианты индивидуальных заданий

1. а) Определить, является ли матрица А магическим квадратом. Входные данные: целое положительное число n , массив чисел А размерности $n \times n$. Матрица является магическим квадратом, когда равны между собой суммы всех строк и суммы всех столбцов. Использовать n или $n+1$ процессов для решения задачи.
б) Заменить наиболее часто встречающийся символ в строке S1 наиболее часто встречающимся символом в строке S2 и наоборот. Входные данные первого приложения: строка символов S1 произвольной длины. Входные данные второго приложения: строка символов S2 произвольной длины.
2. а) В строке символов S заменить каждый a_i символ на b_i символ. Входные данные: целое положительное число n , пары символов (a_1, b_1) , (a_2, b_2) , ... (a_n, b_n) , строка символов S произвольной длины.

Использовать n или $n+1$ процессов для решения задачи.

б) Вычислить скалярное произведение вектора A на вектор B . Входные данные первого приложения: массив чисел A фиксированной размерности. Входные данные второго приложения: массив чисел B , той же размерности, что и массив A .

3. а) Вычислить произведение матриц A и B . Входные данные: целое положительное число n , массивы чисел A и B размерности $n \times n$. Использовать n или $n+1$ процессов для решения задачи.

б) Первое и второе приложение ждут ввода числа от 1 до 10 пользователем. После чего обмениваются полученными значениями. Первое приложение подсчитывает количество совпавших значений, второе приложение находит произведение полученной пары чисел. Ввод чисел пользователями должен быть согласован.

4. а) Вычислить векторное произведение вектора A на вектор B . Входные данные: массивы чисел A и B размерности 3. Использовать не менее трех процессов для решения задачи.

б) Первое приложение заменяет символ a в строке S на символ b . Второе приложение ожидает ввода символа a , третье приложение ожидает ввода символа b . Входные данные первого приложения: строка S произвольной длины. Входные данные второго приложения: символ a . Входные данные третьего приложения: символ b . Время работы приложений не ограничено.

5. а) Определить, совпадает ли хотя бы одна пара – сумма i -й строки и i -го столбца матрицы A . Входные данные: целое положительное число n , массив чисел A размерности $n \times n$. Использовать n или $n+1$ процессов для решения задачи.

б) Первое приложение находит сумму элементов массива A и B , второе приложение находит произведение элементов массива A и B . Входные данные первого приложения: массив чисел A фиксированной размерности. Входные данные второго приложения: массив чисел B той же размерности, что и массив A .

6. а) Определить, совпадает ли хотя бы одна пара – сумма i -й строки и i -го столбца матрицы A . Входные данные: целое положительное число n , массив чисел A размерности $n \times n$. Использовать два процесса для решения задачи.

б) Первое приложение случайным образом определяет число A . Второе приложение ожидает ввода числа B пользователем, третье приложение ожидает ввода числа C . Второе и третье приложения

посылают числа первому, оно определяет, совпадает ли полученное число с числом A , и отправляет назад ответ. Если ответ положительный, пользователю присуждается очко. Игра ведется 15 раундов. Каждый пользователь должен узнавать об успехах другого на каждом раунде. Работа приложений должна быть согласована.

7. а) Вычислить скалярное произведение вектора A на вектор B . Входные данные: целое положительное число n , массивы чисел A и B размерности n , целое положительное число k от 1 до $n/2$. Использовать n/k процессов для решения задачи.

б) Заменить наиболее часто встречающийся символ в строке S символом a_1 . Затем заменить наиболее часто встречающийся символ в строке S , отличный от a_1 , символом a_2 . Продолжать до тех пор, пока работает второе приложение. Входные данные первого приложения: строка символов S произвольной длины. Входные данные второго приложения: символы a_i . Второе приложение, получив a_i символ, выводит получившуюся строку S на экран. Количество символов, которое можно ввести ограничено длиной строки S . Работа приложений должна быть согласована.

8. а) Найти максимальный элемент в матрице A . Входные данные: целые положительные числа n и k , массив чисел A размерности $n \times k$. Использовать n или k процессов для решения задачи.

б) Определить, является ли каждый i -й элемент массива A больше i -го элемента массива B . Входные данные первого приложения: массив чисел A фиксированной размерности. Входные данные второго приложения: массив чисел B той же размерности, что и массив A .

9. а) Проложена дорога, ведущая от города A к городу B , от B к B , от B к Γ , от Γ к D . В каждом городе есть некоторое число пассажиров, желающих поехать в другие города. Но автобусы ходят только между соседними городами, каждый автобус вмещает в себя только 20 пассажиров. Необходимо доставить всех пассажиров в пункты назначения, отображая при этом происходящие изменения. Использовать 4 или 5 процессов для решения задачи.

б) Определить все символы, содержащиеся как в строке S_1 , так и в строке S_2 . Входные данные первого приложения: строка символов S_1 произвольной длины. Входные данные второго приложения: строка символов S_2 произвольной длины.

10. а) Найти индексы i и j , для которых существует наибольшая последовательность $a[i] - a[i+1] + a[i+2] - a[i+3] \dots \pm a[j]$. Входные

данные: целое положительное число n , массив чисел A размерности n .

б) Первое приложение ожидает ввода чисел a , b , c и отправляет их второму приложению, которое находит решение уравнения $ax^2+bx+c=0$, и отправляет результат первому приложению.

11. а) Найти столбец и строку с минимальными суммами в матрице A . Входные данные: целые положительные числа n и k , массив чисел A размерности $n \times k$. Использовать n или k процессов для решения задачи.

б) В первом и втором приложении пользователи вводят массивы чисел 5×5 , содержащие нули и единицы, количество единиц должно быть не менее десяти. На каждом ходе пользователи делают предположение о расположении в массиве другого пользователя единицы – вводят номер строки и номер столбца. Побеждает тот, кто первым найдет единицу в массиве соперника. Работа приложений должна быть согласована.

12. а) Определить, какая сумма элементов массива A является максимальной: сумма всех простых чисел или сумма всех четных чисел. Входные данные: целое положительное число n , массив чисел A размерности n . Использовать два процесса для решения задачи.

б) Два приложения обмениваются случайными тройками чисел (a, b, c) до тех пор, пока две строки не окажутся равными между собой без учета порядка. Работа приложений должна быть согласована.

13. а) В двоичном массиве A определить индексы x_1, y_1, x_2, y_2 с максимальным значением $(x_2-x_1)+(y_2-y_1)$ для которых существует множество одинаковых между собой элементов, заключенных в «прямоугольнике» с верхним левым углом (x_1, y_1) и нижним правым углом (x_2, y_2) .

б) Определить все индексы i , для которых элемент массива A равен элементу массива B . Входные данные первого приложения: массив чисел A фиксированной размерности. Входные данные второго приложения: массив чисел B той же размерности, что и массив A .

14. а) Определить, равны ли между собой суммы двух главных диагоналей в матрице A . Входные данные: целое положительное число n , массив чисел A размерности n . Использовать два процесса для решения задачи.

б) Первое приложение находит все символы, содержащиеся в строке S_1 и не содержащиеся в строке S_2 . Второе приложение находит все символы, содержащиеся в строке S_2 и не содержащиеся в строке S_1 . Входные данные первого приложения: строка символов S_1

произвольной длины. Входные данные второго приложения: строка символов S_2 произвольной длины.

15. а) Задана строка S , содержащая не менее двух слов. Необходимо найти слово, содержащее максимальное количество вхождений символа a . Входные данные: строка S , символ a . Для решения задачи использовать столько процессов, сколько слов в строке.

б) Поменять местами соответственные элементы в массивах A и B , если хотя бы один элемент является простым числом. Входные данные первого приложения: массив чисел A фиксированной размерности. Входные данные второго приложения: массив чисел B той же размерности, что и массив A .

16. а) В матрице A найти строку с максимальным произведением. Входные данные: целое положительное число n , массив чисел A $n \times n$. Использовать n или $n+1$ процессов для решения задачи.

б) Первое приложение обладает банком вопросов и ответов (не менее 10). Второе и третье приложение запускаются игроками. Первое приложение отправляет один и тот же случайный вопрос обоим игрокам и ожидает ответы. Игрок, ответивший правильно, получает очко. Игра ведется в три хода. В завершении игроки узнают результат игры.

17. а) Найти максимальное простое число в массиве A . Входные данные: целое положительное число $n > 4$, массив чисел A размерности n . Использовать четыре процесса для решения задачи.

б) Первое приложение содержит список студентов по предмету – фамилия, оценки. Пользователь, запустив второе приложение, может послать запрос: а) вывести фамилию самого успешного студента (по среднему баллу), б) вывести фамилию самого неуспешного студента (по среднему баллу), в) добавить указанную оценку указанному студенту.

18. а) Найти наиболее часто встречающуюся сумму строки матрицы A . Входные данные: целое положительное число n , массив чисел A размерности $n \times n$. Использовать n или $n+1$ процессов для решения задачи.

б) Первое приложение заменяет максимальный элемент в массиве A минимальным элементом из массива B . Второе приложение заменяет максимальный элемент в массиве B средним арифметическим элементов массива A . Входные данные первого приложения: массив чисел A произвольной размерности. Входные данные второго

приложения: массив чисел В той же размерности, что и массив А.

19. а) Вычислить суммы элементов главной диагонали, элементов, стоящих ниже главной диагонали, и элементов, стоящих выше главной диагонали. Определить минимальную и максимальную сумму. Входные данные: целое положительное число $n > 2$, массив чисел А размерности $n \times n$. Использовать 3 или 4 процесса для решения задачи.

б) Составить строку S3, добавив в нее элементы S1[i] и S2[i], отличные между собой. Входные данные первого приложения: строка символов S1 произвольной длины. Входные данные второго приложения: строка символов S2 произвольной длины.

20. а) Задана строка S, содержащая не менее двух чисел. Необходимо найти наибольшее и наименьшее число. Входные данные: строка S произвольной длины. Для решения задачи использовать столько процессов, сколько чисел записано в строке.

б) В первом и втором приложении игроки бросают два кубика. Очко зачисляется тому игроку, для которого сумма выпавших значений больше, или же игроку с одинаковыми выпавшими значениями. Если оба игрока имеют одинаковые выпавшие им значения, побеждает тот, чья сумма больше. Игра ведется в 8 ходов. На каждом ходе каждый игрок должен получать информацию о своих очках и очках противника. Работа приложений должна быть согласована.

Контрольные вопросы

1. Приведите функции для работы с неименованными каналами, опишите их окружение.
2. Опишите именованные каналы (FIFO - каналы), функции работы с ними.
3. Опишите процедуру открытия файла. Системный вызов `open()`
4. Опишите системные вызовы `read()`, `write()`. Системный вызов `close()`.
5. Опишите особенности поведения вызова `open()` при открытии FIFO
6. Представьте особенности поведения вызовов `read()` и `write()` для `pip'a`
7. Предположим, вам нужно разработать новую компьютерную архитектуру, которая вместо использования прерываний осуществляет аппаратное переключение процессов. Какие сведения необходимы центральному процессору? Опишите

возможное устройство аппаратного переключения процессов.

8. На всех ныне существующих компьютерах хотя бы часть обработчиков прерываний написаны на ассемблере. Почему?
9. Когда в результате прерывания или системного вызова управление передается операционной системе, используется, как правило, область стека ядра, отделенная от стека прерываемого процесса. Почему?
10. Несколько заданий могут быть запущены параллельно и завершить работу быстрее, чем при последовательном запуске. Предположим, что два задания, на каждое из которых требуется 10 мин процессорного времени, запускаются одновременно. Сколько времени пройдет до завершения второго из них, если они будут запущены последовательно? А сколько времени пройдет, если они запущены параллельно? При этом предположим, что на ожидание завершения операций ввода-вывода затрачивается 50% времени.
11. При разветвлении многопоточного процесса возникает проблема при копировании в дочерний процесс всех потоков родительского процесса. Предположим, что один из исходных потоков находится в состоянии ожидания ввода с клавиатуры. Теперь клавиатурного ввода будут ожидать два потока, по одному в каждом процессе. Может ли подобная проблема когда-либо возникнуть в однопоточных процессах?
12. В чем заключается самое большое преимущество от реализации потоков в пользовательском пространстве? А в чем заключается самый серьезный недостаток?

Лабораторная работа 5

СРЕДСТВА SYSTEM V IPC. ОРГАНИЗАЦИЯ РАБОТЫ С РАЗДЕЛЯЕМОЙ ПАМЯТЬЮ В UNIX. ПОНЯТИЕ НИТЕЙ ИСПОЛНЕНИЯ (thread)

Цели и задачи

Изучить общие принципы работы с основными средствами межпроцессной коммуникации. Научиться создавать многопроцессные алгоритмы с общей разделяемой памятью. Познакомиться с легковесными процессами – потоками.

5.1 Средства межпроцессной коммуникации (IPC)

Средствами межпроцессной коммуникации (IPC – Inter-Process Communication) являются сигналы, каналы, сообщения, семафоры, разделяемая память и сокеты. Процессы выполняются в собственном адресном пространстве, они изолированы друг от друга, поэтому необходимы механизмы для взаимодействия процессов, предоставляемые самой операционной системой и, как правило, расположенные в адресном пространстве системы.

Коммуникация между процессами необходима для решения следующих задач:

1. Передача данных от одного процесса к другому.
2. Совместное использование общих данных несколькими процессами.
3. Синхронизация работы процессов.

Преимущества и недостатки потокового обмена данными

Потоковые механизмы достаточно просты в реализации и удобны для использования, но имеют ряд существенных недостатков: Операции чтения и записи не анализируют содержимое передаваемых данных. Процесс, прочитавший 20 байт из потока, не может сказать, были ли они записаны одним процессом или несколькими, записывались ли они за один раз или было, например, выполнено 4 операции записи по 5 байт. Данные в потоке никак не интерпретируются системой. Если требуется какая-либо интерпретация данных, то передающий и принимающий процессы должны заранее согласовать свои действия и уметь осуществлять ее самостоятельно.

Для передачи информации от одного процесса другому требуется, как минимум, две операции копирования данных: первый раз – из адресного пространства передающего процесса в системный буфер, второй раз – из системного буфера в адресное пространство принимающего процесса.

Процессы, обменивающиеся информацией, должны одновременно существовать в вычислительной системе. Нельзя записать информацию в поток с помощью одного процесса, завершить его, а через некоторое время запустить другой процесс и прочитать записанную информацию.

5.2 Понятие о System V IPC

Указанные выше недостатки потоков данных привели к разработке других механизмов передачи информации между процессами. Часть этих механизмов (сообщения, семафоры, разделяемая память), впервые появившихся в UNIX System V и впоследствии перекочевавших оттуда практически во все современные версии операционной системы UNIX, получила общее название System V IPC (IPC – сокращение от interprocess communications). Эти механизмы объединяются в единый пакет, потому что их соответствующие системные вызовы обладают близкими интерфейсами, а в их реализации используются многие общие подпрограммы.

Основные общие свойства всех трех механизмов:

1. Для каждого механизма поддерживается общесистемная таблица, элементы которой описывают все существующие в данный момент части (представители) механизма (конкретные сегменты разделяемой памяти, семафоры или очереди сообщений).
2. Элемент таблицы содержит некоторый числовой ключ, который выбран пользователем именем в качестве представителя соответствующего механизма. Чтобы два или более процесса могли использовать некоторый механизм, они должны заранее договориться об именовании используемого представителя этого механизма.
3. Процесс, желающий начать пользоваться одним из механизмов, обращается к системе с необходимым вызовом, входными параметрами которого является ключ объекта и дополнительные флаги, а ответным параметром является числовой дескриптор, используемый в дальнейших системных вызовах подобно тому, как используется дескриптор файла при работе с файловой системой.
4. Защита доступа к ранее созданным элементам таблицы каждого механизма основывается на тех же принципах, что и защита доступа к файлам.

5.3 Пространство имен. Адресация в System V IPC. Функция `ftok()`

Все средства связи из System V IPC, как и уже рассмотренные `pipe` и `FIFO`, являются средствами связи с непрямой адресацией. Для организации взаимодействия неродственных процессов с помощью средства связи с непрямой адресацией необходимо, чтобы это средство связи имело имя. Отсутствие имен у `pipe`'ов позволяет процессам получать информацию о расположении `pipe`'а в системе и его состоянии только через родственные связи. Наличие ассоциированного имени у `FIFO` – имени специализированного файла в файловой системе – позволяет неродственным процессам получать эту информацию через интерфейс файловой системы. Множество всех возможных имен для объектов какого-либо вида принято называть пространством имен соответствующего вида объектов. Для `FIFO` пространством имен является множество всех допустимых имен файлов в файловой системе. Для всех объектов из System V IPC таким пространством имен является множество значений некоторого целочисленного типа данных – **key_t** – ключа. Причем программисту не позволено напрямую присваивать значение ключа, это значение задается опосредованно: через комбинацию имени какого-либо файла, уже существующего в файловой системе, и небольшого целого числа – например, номера экземпляра средства связи. Такой хитрый способ получения значения ключа вызван двумя соображениями: если разрешить программистам самим присваивать значение ключа для идентификации средств связи, то не исключено, что два программиста случайно воспользуются одним и тем же значением, не подозревая об этом. Тогда их процессы будут несанкционированно взаимодействовать через одно и то же средство коммуникации, что может привести к нестандартному поведению этих процессов. Поэтому основным компонентом значения ключа является преобразованное в числовое значение полное имя некоторого файла, доступ к которому на чтение разрешен процессу. Каждый программист имеет возможность использовать для этой цели свой специфический файл, например исполняемый файл, связанный с одним из взаимодействующих процессов. Преобразование из текстового имени файла в число основывается на расположении указанного файла на жестком диске или ином физическом носителе. Поэтому для образования ключа следует применять файлы, не

меняющие своего положения в течение времени организации взаимодействия процессов; второй компонент значения ключа используется для того, чтобы позволить программисту связать с одним и тем же именем файла более одного экземпляра каждого средства связи. В качестве такого компонента можно задавать порядковый номер соответствующего экземпляра. Получение значения ключа из двух компонентов осуществляется функцией **ftok()**.

Функция для генерации ключа System V IPC

Прототип функции

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

```
#include <sys/ipc.h>
```

```
key_t ftok(char *pathname, char proj);
```

Описание функции

Функция **ftok** служит для преобразования имени существующего файла и небольшого целого числа, например, порядкового номера экземпляра средств связи, в ключ System V IPC.

Параметр **pathname** должен являться указателем на имя существующего файла, доступного для процесса, вызывающего функцию.

Параметр **proj** — это небольшое целое число, характеризующее экземпляр средства связи.

В случае невозможности генерации ключа функция возвращает отрицательное значение, в противном случае она возвращает значение сгенерированного ключа. Тип данных **key_t** обычно представляет собой 32-битовое целое.

5.4 Дескрипторы System V IPC

Информацию о потоках ввода-вывода, с которыми имеет дело текущий процесс, в частности о **pip**'ах и FIFO, операционная система хранит в таблице открытых файлов процесса. Системные вызовы, осуществляющие операции над потоком, используют в качестве параметра индекс элемента таблицы открытых файлов, соответствующего потоку, — файловый дескриптор. Использование файловых дескрипторов для идентификации потоков внутри процесса

позволяет применять к ним уже существующий интерфейс для работы с файлами, но в то же время приводит к автоматическому закрытию потоков при завершении процесса.

Этим, в частности, объясняется один из перечисленных выше недостатков потоковой передачи информации.

При реализации компонентов System V IPC была принята другая концепция. Ядро операционной системы хранит информацию обо всех средствах System V IPC, используемых в системе, вне контекста пользовательских процессов. При создании нового средства связи или получении доступа к уже существующему процесс получает неотрицательное целое число – дескриптор (идентификатор) этого средства связи, которое однозначно идентифицирует его во всей вычислительной системе. Этот дескриптор должен передаваться в качестве параметра всем системным вызовам, осуществляющим дальнейшие операции над соответствующим средством System V IPC.

Подобная концепция позволяет устранить один из самых существенных недостатков, присущих потоковым средствам связи, – требование одновременного существования взаимодействующих процессов, но в то же время требует повышенной осторожности для того, чтобы процесс, получающий информацию, не принял взамен новых старые данные, случайно оставленные в механизме коммуникации.

5.5 Разделяемая память в UNIX.

Системные вызовы shmget(), shmat(), shmdt()

С точки зрения операционной системы, наименее семантически нагруженным средством System V IPC является разделяемая память (shared memory). Операционная система может позволить нескольким процессам совместно использовать некоторую область адресного пространства. Когда процесс А посылает данные другому процессу В через канал, происходят следующие действия: данные копируются из буфера процесса А в буфер ядра, затем эти же данные копируются из буфера ядра в буфер процесса В. Механизм разделяемой памяти позволяет исключить передачу данных через ядро, предоставляя нескольким процессам доступ к одной и той же области памяти – разделяемой памяти.

Все средства связи System V IPC требуют предварительных инициализирующих действий (создания) для организации взаимодействия процессов.

Для получения информации по механизмам ipc, необходимо использовать команду **\$ipcs**.

Для получения информации только по сегментам разделяемой памяти необходимо использовать команду **\$ipcs -m**.

Для создания области разделяемой памяти с определенным ключом или для доступа по ключу к уже существующей области применяется системный вызов **shmget()**. Существует два варианта его использования для создания новой области разделяемой памяти.

Стандартный способ. В качестве значения ключа системному вызову поставляется значение, сформированное функцией **ftok()** для некоторого имени файла и номера экземпляра области разделяемой памяти. В качестве флагов поставляется комбинация прав доступа к создаваемому сегменту и флага **IPC_CREAT**. Если сегмент для данного ключа еще не существует, то система будет пытаться создать его с указанными правами доступа. Если же вдруг он уже существовал, то мы просто получим его дескриптор. Возможно добавление к этой комбинации флагов флага **IPC_EXCL**. Этот флаг гарантирует нормальное завершение системного вызова только в том случае, если сегмент действительно был создан (т. е. ранее он не существовал), если же сегмент существовал, то системный вызов завершится с ошибкой, и значение системной переменной **errno**, описанной в файле **errno.h**, будет установлено в **EEXIST**.

Нестандартный способ. В качестве значения ключа указывается специальное значение **IPC_PRIVATE**. Использование значения **IPC_PRIVATE** всегда приводит к попытке создания нового сегмента разделяемой памяти с заданными правами доступа и с ключом, который не совпадает со значением ключа ни одного из уже существующих сегментов и который не может быть получен с помощью функции **ftok()** ни при одной комбинации ее параметров. Наличие флагов **IPC_CREAT** и **IPC_EXCL** в этом случае игнорируется.

Системный вызов `shmget()`

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size,
int shmflg);
```

Описание системного вызова

Системный вызов **shmget** предназначен для выполнения операции доступа к сегменту разделяемой памяти и, в случае его успешного завершения, возвращает дескриптор **System V IPC** для этого сегмента (целое неотрицательное число, однозначно характеризующее сегмент внутри вычислительной системы и используемое в дальнейшем для других операций с ним).

Параметр **key** является ключом **System V IPC** для сегмента, т. е. фактически его именем из пространства имен **System V IPC**. В качестве значения этого параметра может использоваться значение ключа, полученное с помощью функции **ftok()**, или специальное значение **IPC_PRIVATE**. Использование значения **IPC_PRIVATE** всегда приводит к попытке создания нового сегмента разделяемой памяти с ключом, который не совпадает со значением ключа ни одного из уже существующих сегментов и который не может быть получен с помощью функции **ftok()** ни при одной комбинации ее параметров.

Параметр **size** определяет размер создаваемого или уже существующего сегмента в байтах. Если сегмент с указанным ключом уже существует, но его размер не совпадает с указанным в параметре **size**, констатируется возникновение ошибки.

Параметр **shmflg** – флаги – играет роль только при создании нового сегмента разделяемой памяти и определяет права различных пользователей при доступе к сегменту, а также необходимость создания нового сегмента и поведение системного вызова при попытке создания. Он является некоторой комбинацией (с помощью операции побитовое или – " | ") следующих предопределенных значений и восьмеричных прав доступа:

IPC_CREAT – если сегмента для указанного ключа не существует, он должен быть создан;

IPC_EXCL – применяется совместно с флагом **IPC_CREAT**. При совместном их использовании и существовании сегмента с указанным ключом, доступ к сегменту не производится и констатируется ошибочная ситуация, при этом переменная **errno**, описанная в файле **<errno.h>**, примет значение **EEXIST**;

0400 – разрешено чтение для пользователя, создавшего сегмент;

0200 – разрешена запись для пользователя, создавшего сегмент;

0040 – разрешено чтение для группы пользователя, создавшего сегмент;

0020 – разрешена запись для группы пользователя, создавшего сегмент;

0004 – разрешено чтение для всех остальных пользователей;

0002 – разрешена запись для всех остальных пользователей.

Возвращаемое значение

Системный вызов возвращает значение дескриптора **System V IPC** для сегмента разделяемой памяти при нормальном завершении и значение -1 при возникновении ошибки.

Доступ к созданной области разделяемой памяти в дальнейшем обеспечивается ее дескриптором, который вернет системный вызов **shmget()**. Доступ к уже существующей области также может осуществляться двумя способами: если знать ее ключ, то, используя вызов **shmget()**, можно получить ее дескриптор. В этом случае нельзя указывать в качестве составной части флагов флаг **IPC_EXCL**, а значение ключа, естественно, не может быть **IPC_PRIVATE**. Права доступа игнорируются, а размер области должен совпадать с размером, указанным при ее создании; либо можно воспользоваться тем, что дескриптор **System V IPC** действителен в рамках всей операционной системы, и передать его значение от процесса, создавшего разделяемую память, текущему процессу. При создании разделяемой памяти с помощью значения **IPC_PRIVATE** – это единственно возможный способ.

После получения дескриптора необходимо включить область разделяемой памяти в адресное пространство текущего процесса. Это осуществляется с помощью системного вызова **shmat()**. При нормальном завершении он вернет адрес разделяемой памяти в

адресном пространстве текущего процесса. Дальнейший доступ к этой памяти осуществляется с помощью обычных средств языка программирования.

Системный вызов **shmat()**

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
char *shmat(int shmid, char *shmaddr,
             int shmflg);
```

Описание системного вызова

Системный вызов **shmat** предназначен для включения области разделяемой памяти в адресное пространство текущего процесса. Данное описание не является полным описанием системного вызова. Для полного описания обращайтесь к **UNIX Manual**.

Параметр **shmid** является дескриптором **System V IPC** для сегмента разделяемой памяти, т. е. значением, которое вернул системный вызов **shmget()** при создании сегмента или при его поиске по ключу.

В качестве параметра **shmaddr** в рамках нашего курса мы всегда будем передавать значение **NULL**, позволяя операционной системе самой разместить разделяемую память в адресном пространстве нашего процесса.

Параметр **shmflg** в нашем курсе может принимать только два значения: **0** — для осуществления операций чтения и записи над сегментом и **SHM_RDONLY** — если мы хотим только читать из него. При этом процесс должен иметь соответствующие права доступа к сегменту.

Возвращаемое значение

Системный вызов возвращает адрес сегмента разделяемой памяти в адресном пространстве процесса при нормальном завершении и значение **-1** при возникновении ошибки.

После окончания использования разделяемой памяти процесс может уменьшить размер своего адресного пространства, исключив из него эту область с помощью системного вызова **shmdt()**. В качестве параметра системный вызов **shmdt()** требует адрес начала области

разделяемой памяти в адресном пространстве процесса, т. е. значение, которое вернул системный вызов **shmat()**, поэтому данное значение следует сохранять на протяжении всего времени использования разделяемой памяти.

Системный вызов shmdt()

Прототип системного вызова

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

```
#include <sys/ipc.h>
```

```
#include <sys/shm.h>
```

```
int shmdt(char *shmaddr);
```

Описание системного вызова

Системный вызов **shmdt** предназначен для исключения области разделяемой памяти из адресного пространства текущего процесса.

Параметр **shmaddr** является адресом сегмента разделяемой памяти, т. е. значением, которое вернул системный вызов **shmat()**.

Возвращаемое значение

Системный вызов возвращает значение **0** при нормальном завершении и значение **-1** при возникновении ошибки.

Для иллюстрации использования разделяемой памяти рассмотрим две взаимодействующие программы.

Программа 5-01a для иллюстрации работы с разделяемой памятью.

Организация разделяемой памяти для массива из трех целых чисел. Первый элемент массива является счетчиком числа запусков программы 1a, второй элемент массива – счетчиком числа запусков программы 1b, третий элемент массива – счетчиком числа запусков обеих программ

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

```
#include <sys/ipc.h>
```

```
#include <sys/shm.h>
```

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

```
#include <errno.h>
```

```

#include <iostream>
using namespace std;
int main()
{
    int *array; /* Указатель на разделяемую память */
    int shmid; /* IPC дескриптор для области разделяемой памяти */
    int n = 1; /* Флаг необходимости инициализации элементов
массива */
    char pathname[] = "Имя файла"; /* Имя файла, используемое для
генерации ключа. Файл с таким именем должен существовать в
текущей директории */
    key_t key; /* IPC ключ */
    /* Генерирование IPC ключа из имени файла для генерации ключа в
текущей директории и номера экземпляра области разделяемой
памяти 0 */
    if((key = ftok(pathname,0)) < 0)
    {
        cout<<"Can't generate key"<<endl;
    }
    /* Эксклюзивное создание разделяемой памяти для
сгенерированного ключа, т. е. если для этого ключа она уже
существует, системный вызов вернет отрицательное значение.
Размер памяти определяем как размер массива из трех целых
переменных, права доступа 0666 – чтение и запись разрешены для
всех */
    if((shmid = shmget(key, 3*sizeof(int), 0666|IPC_CREAT|IPC_EXCL)) <
0)
    {
        /* В случае ошибки необходимо определить: возникла ли она из-за
того, что сегмент разделяемой памяти уже существует, или по
другой причине */
        if(errno != EEXIST)
        {
            /* Если по другой причине – прекращение работы */
            cout<<"Can't create shared memory"<<endl;
        }
    }
    else
    {

```


/ Если из-за того, что разделяемая память уже существует, то необходимо получить ее IPC дескриптор и, в случае удачи, сброс флага необходимости инициализации элементов массива */*

```
if((shmid = shmget(key, 3*sizeof(int), 0)) < 0)
{
    cout<<"Can't find shared memory"<<endl;

    }
    n = 0;
}
```

*/*Отображение разделяемой памяти в адресное пространство текущего процесса. Обратите внимание на то, что для правильного сравнения мы явно преобразовываем значение -1 к указателю на целое.*/*

```
if((array = (int *)shmat(shmid, NULL, 0)) == (int *)(-1))
{
    cout<<"Can't attach shared memory"<<endl;
}
```

/ В зависимости от значения флага new либо инициализация массива, либо увеличение соответствующих счетчиков */*

```
if(n==1)
{
    array[0] = 1;
    array[1] = 0;
    array[2] = 1;
}
else
{
    array[0] += 1;
    array[2] += 1;
}
```

/ Печать новых значений счетчиков, удаление разделяемой памяти из адресного пространства текущего процесса и завершение работы */*

```

cout<<"Program 1 was spawn "<< array[0] <<"times, program 2 – "<<
array[1]<< " times, total –"<< array[2]<< " times"<<endl;
    if(shmdt(array) < 0)
    {
cout<<"Can't detach shared memory"<<endl;

    }
    return 0;
}

```

Листинг 5-01a. Программа 5-01a для иллюстрации работы с разделяемой памятью

Программа 5-01b для иллюстрации работы с разделяемой памятью.

Организация разделяемой памяти для массива из трех целых чисел. Первый элемент массива является счетчиком числа запусков программы 1a, второй элемент массива – счетчиком числа запусков программы 1b, третий элемент массива – счетчиком числа запусков обеих программ

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <errno.h>
#include <iostream>
using namespace std;
int main()
{
    int *array; /* Указатель на разделяемую память */
    int shmid; /* IPC дескриптор для области разделяемой памяти */
    int n = 1; /* Флаг необходимости инициализации элементов
массива */
    char pathname[] = "Имя файла"; /* Имя файла, используемое для
генерации ключа. Файл с таким именем должен существовать в
текущей директории */
    key_t key; /* IPC ключ */

```

/ Генерирование IPC ключа из имени файла для генерации ключа в текущей директории и номера экземпляра области разделяемой памяти 0 */*

if((key = ftok(pathname,0)) < 0)

{

cout<<"Can't generate key"<<endl;

}

/ Эксклюзивное создание разделяемой памяти для сгенерированного ключа, т. е. если для этого ключа она уже существует, системный вызов вернет отрицательное значение. Размер памяти определяем как размер массива из трех целых переменных, права доступа 0666 – чтение и запись разрешены для всех */*

*if((shmid = shmget(key, 3*sizeof(int), 0666/IPC_CREAT/IPC_EXCL)) < 0)*

{

/ В случае ошибки необходимо определить: возникла ли она из-за того, что сегмент разделяемой памяти уже существует, или по другой причине */*

if(errno != EEXIST)

{

/ Если по другой причине – прекращение работы */*

cout<<"Can't create shared memory"<<endl;

}

else

{

/ Если из-за того, что разделяемая память уже существует, то необходимо получить ее IPC дескриптор и, в случае удачи, сброс флага необходимости инициализации элементов массива */*

*if((shmid = shmget(key, 3*sizeof(int), 0)) < 0)*

{

cout<<"Can't find shared memory"<<endl;

}

n = 0;

}

}

*/*Отображение разделяемой памяти в адресное пространство текущего процесса. Обратите внимание на то, что для правильного сравнения мы явно преобразовываем значение -1 к указателю на целое.*/*

```
    if((array = (int *)shmat(shmid, NULL, 0)) == (int *)(-1))
    {
        cout<<"Can't attach shared memory"<<endl;
    }
```

/ В зависимости от значения флага new либо инициализация массива, либо увеличение соответствующих счетчиков */*

```
    if(n==1)
    {
        array[0] = 0;
        array[1] = 1;
        array[2] = 1;
    }
    else
    {
        array[1] += 1;
        array[2] += 1;
    }
```

/ Печать новых значений счетчиков, удаление разделяемой памяти из адресного пространства текущего процесса и завершение работы */*

```
    cout<< "Program 1 was spawn " << array[0] << "times, program 2 –"
    << array[1] << " times, total –" << array[2] << "times" << endl;
    if(shmdt(array) < 0)
    {
        cout<<"Can't detach shared memory"<<endl;
    }
    return 0;
}
```

Листинг 5-01б. Программа для иллюстрации работы с разделяемой памятью

Эти программы очень похожи друг на друга и используют разделяемую память для хранения числа запусков каждой из программ и их суммы. В разделяемой памяти размещается массив из трех целых чисел. Первый элемент массива используется как счетчик для программы 1, второй элемент – для программы 2, третий элемент – для обеих программ суммарно. Дополнительный нюанс в программах возникает из-за необходимости инициализации элементов массива при создании разделяемой памяти. Для этого нужно, чтобы программы могли различать случай, когда они создали ее, и случай, когда она уже существовала. Различия добиваемся тем, что вначале используем системный вызов `shmget()` с флагами `IPC_CREAT` и `IPC_EXCL`. Если вызов завершается нормально, то разделяемая память создана. Если вызов завершается с констатацией ошибки и значение переменной `errno` равняется `EEXIST`, то, значит, разделяемая память уже существует и можно получить ее `IPC` дескриптор, применяя тот же самый вызов с нулевым значением флагов.

Приведем пример работы разделяемой памяти.

Процесс-родитель создает сегмент разделяемой памяти и порождает дочерний процесс, процесс-родитель находит сумму первой половины элементов массива, процесс-потомок находит сумму второй половины элементов массива и записывает вычисленную сумму в разделяемую память. Родительский процесс дожидается окончания работы потомка и выводит окончательный результат – сумму всех элементов массива.

При такой организации алгоритма операции с разделяемой памятью остаются безопасными и при отсутствии семафоров.

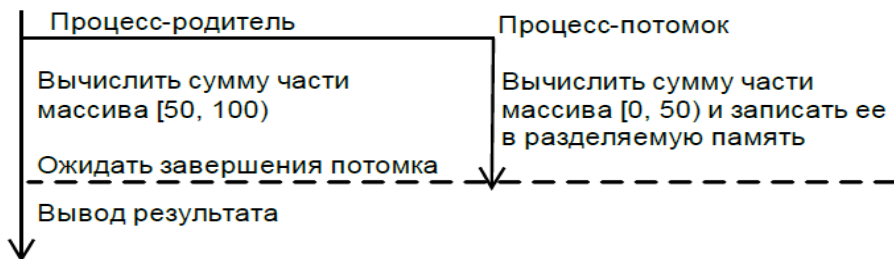


Рис. 5.1. Схема процессов

Программа 5-02. Пример работы разделяемой памяти

```
#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <iostream>
using namespace std;
int A[100] ;
struct mymem
{
    int sum;
}
*mem_sum;
int main()
{
    int shmid = shmget(IPC_PRIVATE, 2, IPC_CREAT|0666) ;
    if (shmid < 0 )
    {
        cout<<"error"<<endl;
        return 0 ;
    }
    mem_sum = (mymem *)shmat(shmid,NULL,0) ;
    A[2]=5; //инициализация массива
    A[56]=90; //взял три числа в разных частях массива
    A[4]=6; //в сумме должно дать 101
    int pid, sum=0 ;
    pid = fork() ;
    if ( pid == 0 )
    {
        for (int i=0 ; i<50 ; i++) sum+=A[i] ;
        mem_sum->sum=sum ;
    }
    if ( pid != 0 )
    {
        for (int i=50 ; i<100 ; i++) sum+=A[i] ;
        wait(NULL);
    }
}
```

```
cout<<"Calculate = " << sum+mem_sum->sum<<endl;
}
return 1;
}
```

Листинг 5-02. Пример работы разделяемой памяти

5.6 Команды **ipcs** и **ipcrm**

Команда **ipcs** выдает информацию обо всех средствах System V IPC, существующих в системе, для которых пользователь обладает правами на чтение: областях разделяемой памяти, семафорах и очередях сообщений.

Команда **ipcs**

Синтаксис команды

ipcs [-asmq] [-tclup]

ipcs [-smq] -i id

ipcs -h

Описание команды

Команда **ipcs** предназначена для получения информации о средствах **System V IPC**, к которым пользователь имеет право доступа на чтение.

Опция **-i** позволяет указать идентификатор ресурсов. Будет выдаваться только информация для ресурсов, имеющих этот идентификатор.

Виды IPC ресурсов могут быть заданы с помощью следующих опций: **-s** для семафоров;

-m для сегментов разделяемой памяти; **-q** для очередей сообщений; **-a** для всех ресурсов (по умолчанию).

Опции **[-tclup]** используются для изменения состава выходной информации. По умолчанию для каждого средства выводятся его ключ, идентификатор **IPC**, идентификатор владельца, права доступа и ряд других характеристик. Применение опций позволяет вывести:

-t времена совершения последних операций над средствами **IPC**;

-p идентификаторы процесса, создавшего ресурс, и процесса, совершившего над ним последнюю операцию;

-c идентификаторы пользователя и группы для создателя ресурса и его собственника;

-l системные ограничения для средств **System V IPC**;

-u общее состояние **IPC** ресурсов в системе.

Опция **-h** используется для получения краткой справочной информации.

Из всего многообразия выводимой информации нас будут интересовать только **IPC** идентификаторы для средств, созданных вами. Эти идентификаторы будут использоваться в команде **ipcrm**, позволяющей удалить необходимый ресурс из системы. Для удаления сегмента разделяемой памяти эта команда имеет вид:

ipcrm shm <IPC идентификатор>

Команда ipcrm

Синтаксис команды

ipcrm [shm / msg / sem] id

Описание команды

Команда **ipcrm** предназначена для удаления ресурса **System V IPC** из операционной системы. Параметр **id** задает **IPC** идентификатор для удаляемого ресурса, параметр **shm** используется для сегментов разделяемой памяти, параметр **msg** – для очередей сообщений, параметр **sem** – для семафоров.

Если поведение программ, использующих средства **System V IPC**, базируется на предположении, что эти средства были созданы при их работе, необходимо перед их запуском удалять уже существующие ресурсы.

5.7 Использование системного вызова shmctl() для освобождения ресурса

Этот системный вызов позволяет полностью ликвидировать область разделяемой памяти в операционной системе по заданному дескриптору средства **IPC**, если есть для этого необходимые полномочия.

Системный вызов **shmctl()**

Прототип системного вызова

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

int shmctl(int shmid, int cmd,
struct shmid_ds *buf);

Описание системного вызова

Системный вызов **shmctl** предназначен для получения информации об области разделяемой памяти, изменения ее атрибутов и удаления из системы. Данное описание не является полным описанием системного вызова. Для изучения полного описания обращайтесь к **UNIX Manual**. В данном курсе системный вызов **shmctl** будет использован только для удаления области разделяемой памяти из системы. Параметр **shmid** является дескриптором **System V IPC** для сегмента разделяемой памяти, т. е. значением, которое вернул системный вызов **shmget()** при создании сегмента или при его поиске по ключу.

В качестве параметра **cmd** в рамках данного курса всегда будет передаваться значение **IPC_RMID** – команду для удаления сегмента разделяемой памяти с заданным идентификатором. Параметр **buf** для этой команды не используется, поэтому всегда необходимо будет подставлять туда значение **NULL**.

Возвращаемое значение

Системный вызов возвращает значение **0** при нормальном завершении и значение **-1** при возникновении ошибки.

Разделяемая память и системные вызовы fork(), exec() и функция exit(). При выполнении системного вызова **fork()** все области разделяемой памяти, размещенные в адресном пространстве процесса, наследуются порожденным процессом.

При выполнении системных вызовов **exec()** и функции **exit()** все области разделяемой памяти, размещенные в адресном пространстве процесса, исключаются из его адресного пространства, но продолжают существовать в операционной системе.

5.8 Потоки

Потоки предоставляют возможность проведения параллельных или псевдопараллельных, в случае одного процессора, вычислений. Потоки могут порождаться во время работы программы, процесса или другого потока. Основное отличие потоков от процессов заключается в том, что различные потоки имеют различные пути выполнения, но при этом пользуются общей памятью. Таким образом, несколько порожденных в программе потоков могут пользоваться глобальными переменными, и любое изменение данных одним потоком будет доступно и для всех остальных. Путь выполнения потока задается при его создании указанием его стартовой функции, созданный поток начинает выполнять команды этой функции и завершается, когда происходит возврат из функции. Любой поток завершается по окончании работы создавшего его процесса. При создании потока, кроме стартовой функции, ему присуждается буфер для стека, определяемый программистом. Если поток в процессе своей работы превысит размерность стека, выделенного ему программистом, он будет уничтожен системой. Потоки обладают общей памятью, операции с которой также должны защищаться семафорами. Для создания потока используется следующая функция (заголовочный файл - **sched.h**).

int clone (имя стартовой функции, **void *stack**, **int flags**, **void *arg**).

Функция создает процесс или поток, выполняющий стартовую функцию, стек нового процесса/потока будет храниться в **stack**, параметр **arg** определяет входной параметр стартовой функции. Стартовая функция должна иметь такой прототип:

int <имя функции>(void *<имя параметра>)

Параметр **flags** может принимать следующие значения.

CLONE_VM – если флаг установлен, создается потомок, обладающий общей памятью с процессом-родителем (поток), если флаг не установлен, создается потомок, которому не доступна память процесса-родителя (процесс).

CLONE_FS - если флаг установлен, потомок обладает той же информацией о файловой системе, что и родитель.

CLONE_FILES - если флаг установлен, потомок обладает

теми же файловыми дескрипторами, что и родитель.

CLONE_SIGHAND – если флаг установлен, потомок обладает той же таблицей обработчиков сигналов, что и родитель.

CLONE_VFORK - если флаг установлен, процесс-родитель будет приостановлен до того момента, пока не завершится созданный им потомок.

Если флаги **CLONE_FS**, **CLONE_FILES**, **CLONE_SIGHAND**, **CLONE_VM** не установлены, потомок при создании получает копию соответствующих флагу данных от процесса-родителя. Следует понимать, что копия данных дублирует информацию от родительского процесса в момент создания потомка, но копия и данные, с которых она получена, занимают различные области памяти. Следовательно, изменение данных в процессе работы родителя неизвестно для потомка и наоборот.

Примеры создания потоков

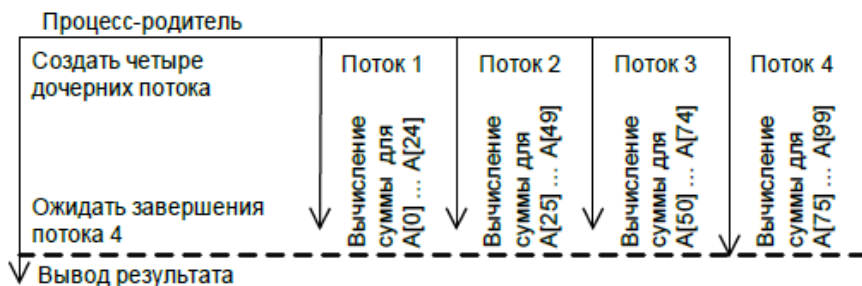


Рис. 5.2. Схема потоков

Программа 5-03. Процесс-родитель создает поток, выполняющий функцию func

```
#include <stdio.h>
#include <sched.h>
#include <unistd.h>
#include <iostream>
using namespace std;
char stack[1000] ; //для хранения стека потока
int func(void * param) //стартовая функция потока
```

```

{
cout<< "Запустился поток"<<endl;
}
int main()
{
clone(func,(void*)(stack+1000-1), CLONE_VM, NULL) ; //создать поток
sleep(1); //блокировка процесса-родителя, чтобы поток успел
выполниться
return 1;
}

```

Листинг 5-03. Процесс-родитель создает поток, выполняющий функцию *func*.

Программа 5-04. Процесс-родитель создает четыре потока, вычисляющих сумму элементов определенной части массива.

Созданные потоки выполняют функцию *func*, получая в качестве параметра индивидуальное целое число от 0 до 3, с помощью которого определяются границы вычисления массива каждым потоком.

```

#include <stdio.h>
#include <sched.h>
#include <unistd.h>
#include <iostream>
#define NUMSTACK 5000 //объем стека для отдельного потока
using namespace std;
int A[100] ; //массив, сумма элементов которого вычисляется
процессами
int SUM=0 ; //для записи общей суммы
char stack[4][ NUMSTACK] ; ///для хранения стека четырех потоков
int func(void *param) //стартовая функция потоков
{
int i, sum = 0 ; //для суммирования элементов
//индекс массива, с которого начинается суммирование
int p = *(int *)param ;
p=p*25 ;
for (i=p ; i<p+25 ; i++) sum+=A[i] ; //вычисление суммы части

```

```

элементов массива
SUM+=sum ; //добавление вычисленного результата в общую
переменную
return 1 ;
}
int main()
{
// тут должна быть инициализация элементов массива A
int param[4] ; //для хранения параметров потоков
for (int i=0 ; i<3 ; i++) //создание трех потоков
{
param[i]=i; //каждому потоку передается уникальное число
char *tostack=stack[i]; //получить указатель на часть массива-стека
потоков
//создать поток со стартовой функцией func
//первый поток получает в качестве параметра 0, второй – 1,
третий - 2
clone(func,(void*)( tostack+ NUMSTACK -1),CLONE_VM, (void
*)(param+i)) ;
}
param[3]=3;
char *tostack=stack[3];
//создание четвертого потока, указание процессу дождаться его
завершения
clone(func,(void*)( tostack+ NUMSTACK -
1),CLONE_VM/CLONE_VFORK, (void *) (param+3));
sleep (1);
cout<<"Результат = "<< SUM <<endl;
return 1;
}

```

Листинг 5-04. Процесс-родитель создает четыре потока, вычисляющих сумму элементов определенной части массива.

5.9 Понятие о нити исполнения (thread) в UNIX.

Идентификатор нити исполнения. Функция pthread_self()

Во многих современных операционных системах существует расширенная реализация понятия процесс, когда процесс представляет собой совокупность выделенных ему ресурсов и набора нитей

исполнения. Нити процесса разделяют его программный код, глобальные переменные и системные ресурсы, но каждая нить имеет собственный программный счетчик, свое содержимое регистров и свой стек. Поскольку глобальные переменные у нитей исполнения являются общими, они могут использовать их как элементы разделяемой памяти, не прибегая к механизму, описанному выше. В различных версиях операционной системы **UNIX** существуют различные интерфейсы, обеспечивающие работу с нитями исполнения. Нити исполнения, удовлетворяющие стандарту **POSIX**, принято называть **POSIX thread**'ами или, кратко, **pthread**'ами.

Операционная система Linux не полностью поддерживает нити исполнения на уровне ядра системы. При создании нового **thread**'а запускается новый традиционный процесс, разделяющий с родительским традиционным процессом его ресурсы, программный код и данные, расположенные вне стека, т. е. фактически действительно создается новый **thread**, но ядро не умеет определять, что эти **thread**'ы являются составными частями одного целого. Это «знает» только специальный процесс-координатор, работающий на пользовательском уровне и стартующий при первом вызове функций, обеспечивающих **POSIX** интерфейс для нитей исполнения. Каждая нить исполнения, как и процесс, имеет в системе уникальный номер – идентификатор **thread**'а. Поскольку традиционный процесс в концепции нитей исполнения трактуется как процесс, содержащий единственную нить исполнения, можно узнать идентификатор этой нити и для любого обычного процесса. Для этого используется функция **pthread_self()**. Нить исполнения, создаваемую при рождении нового процесса, принято называть начальной или главной нитью исполнения этого процесса.

Функция **pthread_self()**

Прототип функции

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

```
pthread_t pthread_self(void);
```

Описание функции

Функция **pthread_self** возвращает идентификатор текущей нити исполнения.

Тип данных **pthread_t** является синонимом для одного из целочисленных типов языка Си.

Создание и завершение thread'a. Функции `pthread_create()`, `pthread_exit()`, `pthread_join()`

Нити исполнения, как и традиционные процессы, могут порождать нити-потомки, правда, только внутри своего процесса. Каждый будущий thread внутри программы должен представлять собой функцию с прототипом:

```
void *thread(void *arg);
```

Параметр **arg** передается этой функции при создании thread'a и может, до некоторой степени, рассматриваться как аналог параметров функции **main()**. Возвращаемое функцией значение может интерпретироваться как аналог информации, которую родительский процесс может получить после завершения процесса-ребенка. Для создания новой нити исполнения применяется функция **pthread_create()**.

Функция для создания нити исполнения

Прототип функции

```
#include <pthread.h>  
int pthread_create(pthread_t *thread,  
    pthread_attr_t *attr,  
    void *(*start_routine)(void *),  
    void *arg);
```

Описание функции

Функция **pthread_create** служит для создания новой нити исполнения (thread'a) внутри текущего процесса.

Новый thread будет выполнять функцию **start_routine** с прототипом

```
void *start_routine(void *)
```

передавая ей в качестве аргумента параметр **arg**. Если требуется передать более одного параметра, они собираются в структуру; передается адрес этой структуры. Значение, возвращаемое функцией

start_routine, не должно указывать на динамический объект данного **thread'a**. Параметр **attr** служит для задания различных атрибутов создаваемого **thread'a**.

Возвращаемые значения

При удачном завершении функция возвращает значение **0** и помещает идентификатор новой нити исполнения по адресу, на который указывает параметр **thread**. В случае ошибки возвращается положительное значение (а не отрицательное, как в большинстве системных вызовов и функций!), которое определяет код ошибки, описанный в файле **<errno.h>**. Значение системной переменной **errno** при этом не устанавливается. Результатом выполнения этой функции является появление в системе новой нити исполнения, которая будет выполнять функцию, ассоциированную со **thread'ом**, передав ей специфицированный параметр, параллельно с уже существовавшими нитями исполнения процесса.

Созданный **thread** может завершить свою деятельность тремя способами: С помощью выполнения функции **pthread_exit()**. Функция никогда не возвращается в вызвавшую ее нить исполнения. Объект, на который указывает параметр этой функции, может быть изучен в другой нити исполнения, например, в породившей завершившийся **thread**. Этот параметр, следовательно, должен указывать на объект, не являющийся локальным для завершившегося **thread'a**, например, на статическую переменную; С помощью возврата из функции, ассоциированной с нитью исполнения. Объект, на который указывает адрес, возвращаемый функцией, как и в предыдущем случае, может быть изучен в другой нити исполнения, например, в породившей завершившийся **thread**, и должен указывать на объект, не являющийся локальным для завершившегося **thread'a**; Если в процессе выполняется возврат из функции **main()** или где-либо в процессе (в любой нити исполнения) осуществляется вызов функции **exit()**, это приводит к завершению всех **thread'ов** процесса.

Функция для завершения нити исполнения

Прототип функции

```
#include <pthread.h>
void pthread_exit(void *status);
```


Описание функции

Функция **pthread_exit** служит для завершения нити исполнения (**thread**) текущего процесса.

Функция никогда не возвращается в вызвавший ее **thread**. Объект, на который указывает параметр **status**, может быть впоследствии изучен в другой нити исполнения, например, в нити, породившей завершившуюся нить. Поэтому он не должен указывать на динамический объект завершившегося **thread'a**.

Одним из вариантов получения адреса, возвращаемого завершившимся **thread'ом**, с одновременным ожиданием его завершения является использование функции **pthread_join()**. Нить исполнения, вызвавшая эту функцию, переходит в состояние «ожидание» до завершения заданного **thread'a**. Функция позволяет также получить указатель, который вернул завершившийся **thread** в операционную систему.

Функция pthread_join()

Прототип функции

#include <pthread.h>

int pthread_join (pthread_t thread,

*void **status_addr);*

Описание функции

Функция **pthread_join** блокирует работу вызвавшей ее нити исполнения до завершения **thread'a** с идентификатором **thread**. После разблокирования в указатель, расположенный по адресу **status_addr**, заносится адрес, который вернул завершившийся **thread** либо при выходе из ассоциированной с ним функции, либо при выполнении функции **pthread_exit()**. Если не интересует, что вернула нить исполнения, в качестве этого параметра можно использовать значение **NULL**.

Возвращаемые значения

Функция возвращает значение **0** при успешном завершении. В случае ошибки возвращается положительное значение (а не отрицательное, как в большинстве системных вызовов и функций!), которое определяет код ошибки, описанный в файле **<errno.h>**. Значение

системной переменной **errno** при этом не устанавливается.

Для иллюстрации вышесказанного рассмотрим программу, в которой работают две нити исполнения.

Программа 5-05 для иллюстрации работы двух нитей исполнения.

Каждая нить исполнения просто увеличивает на 1 разделяемую переменную a.

```
#include <pthread.h>
#include <stdio.h>
#include <iostream>
using namespace std;
int a = 0;
/* Переменная a является глобальной статической для всей
программы, поэтому она будет разделяться обеими нитями
исполнения. */
/* Ниже следует текст функции, которая будет ассоциирована со 2-
м thread'ом */

void *mythread(void *dummy)

/* Параметр dummy в функции не используется и присутствует
только для совместимости типов данных. По той же причине
функция возвращает значение void, хотя это никак не используется в
программе. */
{
    pthread_t mythid; /* Для идентификатора нити исполнения */

    /* Переменная mythid является динамической локальной переменной
функции mythread(),
т. е. помещается в стеке и, следовательно, не разделяется нитями
исполнения. */

    /* Запрос идентификатор thread'a */
```

```

    mythread = pthread_self();
    a = a+1;
    cout<<"Thread" << mythread<< endl;
    cout<< "Calculation result = "<< a << endl;
    return NULL;
}
/* Функция main() – она же ассоциированная функция главного
thread'a */
int main()
{
    pthread_t thid, mythread;
    int result;
    /* Создание новой нити исполнения, ассоциированной с функцией
    mythread(). Передача ей в качестве параметра значение NULL. В
    случае удачи в переменную thid занесется идентификатор нового
    thread'a. Если возникнет ошибка, завершение работы. */
    result = pthread_create( &thid, (pthread_attr_t *)NULL, mythread,
    NULL);
    if(result != 0)
    {
        cout<<"Error on thread create, return value = " << result << endl;
    }
    cout<<"Thread created, thid =" << thid <<endl;
    /* Запрос идентификатора главного thread'a */
    mythread = pthread_self();
    a = a+1;
    cout<<"Thread" << mythread<< endl;
    cout<< "Calculation result = "<< a << endl;
    /* Ожидание завершения порожденного thread'a, не интересуясь,
    какое значение он вернет. Если не выполнить вызов этой функции, то
    возможна ситуация, когда завершится функция main() до того, как
    выполнится порожденный thread, что автоматически повлечет за
    собой его завершение, исказив результаты. */
    pthread_join(thid, (void **)NULL);
    return 0;
}

```

Листинг 5-05. Программа для иллюстрации работы двух нитей исполнения.

Для сборки исполняемого файла при работе редактора связей необходимо явно подключить библиотеку функций для работы с pthread'ами, которая не подключается автоматически. Это делается с помощью добавления к команде компиляции и редактирования связей параметра -lpthread – подключить библиотеку pthread .

Необходимость синхронизации процессов и нитей исполнения, использующих общую память.

Все рассмотренные примеры являются не совсем корректными. В большинстве случаев они работают правильно, однако возможны ситуации, когда совместная деятельность этих процессов или нитей исполнения приводит к неверным и неожиданным результатам. Это связано с тем, что любые неатомарные операции, связанные с изменением содержимого разделяемой памяти, представляют собой критическую секцию процесса или нити исполнения.

При одновременном существовании двух процессов в операционной системе может возникнуть следующая последовательность выполнения операций во времени:

...

Процесс 1: array[0] += 1;

Процесс 2: array[1] += 1;

Процесс 1: array[2] += 1;

*Процесс 1: cout<<"Program 1 was spawn" << array[0] << "times,
program 2 –" <<array[0] << "times, total –" << array[0]<<
"times"<< endl;*

Тогда печать будет давать неправильные результаты. Естественно, что воспроизвести подобную последовательность действий практически нереально. Невозможно подобрать необходимое время старта процессов и степень загруженности вычислительной системы. Но можно смоделировать эту ситуацию, добавив в обе программы достаточно длительные пустые циклы перед оператором array[2] += 1; Это сделано в последующих программах.

Порядок выполнения лабораторной работы

- 1.** Наберите программы листингов 5-01a и 5-01b, сохраните под именами 05-1a.c и 05-1b.c соответственно, откомпилируйте их и запустите несколько раз. Проанализируйте полученные результаты.
- 2.** Для закрепления знаний по разделяемой памяти напишите две программы, осуществляющие взаимодействие через разделяемую память. Первая программа должна создавать сегмент разделяемой памяти и копировать туда собственный исходный текст, вторая программа должна брать оттуда этот текст, печатать его на экране и удалять сегмент разделяемой памяти из системы.
- 3.** Разработать алгоритм решения задания 5-02 с учетом разделения вычислений между несколькими процессами. Для обмена информацией между процессами использовать разделяемую память. Реализуйте алгоритм решения задания 5-02 и протестируйте на нескольких примерах.
- 4.** Наберите текст листинга 5-03, откомпилируйте эту программу и запустите на исполнение. (прогон программы с 2 нитями). Модифицируйте, добавив 3 нить.
- 5.** Наберите программы, сохраните под именами 05-3a.c и 05-3b.c соответственно, откомпилируйте их и запустите любую из них один раз для создания и инициализации разделяемой памяти. Затем запустите другую и, пока она находится в цикле, запустите, например, с другого виртуального терминала снова первую программу. Вы получите неожиданный результат: количество запусков по отдельности не будет соответствовать количеству запусков вместе.
- 6.** Разработать алгоритм решения индивидуального задания б с учетом разделения вычислений между несколькими потоками.
- 7.** Реализовать алгоритм решения задания б и протестировать на нескольких примерах.
- 8.** Для обоих созданных приложений посмотреть в динамике работу разделяемой памяти, используя команду `ipcs -m`.
- 9.** Модифицируйте программы из этого раздела для корректной работы с помощью алгоритма Петерсона.

Варианты индивидуальных заданий

1. а) Задана строка S , содержащая не менее двух слов. Необходимо найти среди слов, палиндром максимальной длины. Входные данные: строка S . Для решения задачи использовать столько процессов, сколько слов в строке. Палиндромом является фраза или слово, одинаково читаемые как слева направо, так и справа налево, пример – поп.
б) Даны результаты сдачи экзамена по группам. Создать многопоточное приложение, вычисляющее количество двоечников и отличников в каждой группе.
2. а) Найти максимальный M и минимальный элемент m массива A и составить множество чисел, лежащих в интервале (m, M) и не содержащихся в массиве A . Входные данные: целое положительное число n , массив чисел A размерности n .
б) Дан список студентов по группам. Создать многопоточное приложение для определения количества студентов с фамилией Иванов.
3. а) Определить какая сумма больше: сумма всех положительных чисел, стоящих выше главной диагонали и ниже второй главной диагонали, или сумма модулей всех отрицательных чисел, стоящих выше второй главной диагонали и ниже главной диагонали. Входные данные: целое положительное число n , массив чисел A размерности $n \times n$.
б) Охранное агентство разработало новую систему управления электронными замками. Для открытия двери клиент обязан произнести произвольную фразу из 10 слов. В этой фразе должно встречаться заранее оговоренное слово, причем только два раза. Создать многопоточное приложение, управляющее замком.
4. а) Найти среднее арифметическое всех «особых» элементов матрицы A . Будем считать, элемент особым, если он больше суммы всех остальных элементов, стоящих в том же столбце. Входные данные: целое положительное число n , массив чисел A размерности $n \times n$. Использовать n или $n+1$ процессов для решения задачи.
б) Даны результаты сдачи экзамена по группам. Создать многопоточное приложение, вычисляющее общий средний балл и средний балл для каждой группы.
5. а) Найти сумму индекса строки с максимальным элементом на

главной диагонали с индексом столбца с минимальным элементом на второй главной диагонали. Входные данные: целое положительное число n , массив чисел A размерности $n \times n$.

б) Командиру воинской части полковнику Кузнецову требуется перемножить два секретных числа. Полковник Кузнецов вызывает дежурного по части лейтенанта Смирнова и требует предоставить ему ответ. Лейтенант Смирнов будит старшего по караулу сержанта Петрова и приказывает ему предоставить ответ. Сержант Петров вызывает к себе рядового Иванова и поручает ему ответственное задание по определению произведения. Рядовой Иванов успешно справляется с поставленной задачей, и ответ передается полковнику Кузнецову. Создать многопоточное приложение, в котором все военнослужащие от полковника до рядового моделируются потоками одного вида.

6. а) Составить строку из максимальных и минимальных элементов строк матрицы A . Порядок элементов в строке не важен. Входные данные: целое положительное число n , целое положительное число k , массив чисел от 0 до 9 A размерности $n \times k$. Использовать n или $n+1$ процессов для решения задачи.

б) Изготовление знаменитого самурайского меча катаны происходит в три этапа. Младший ученик мастера выковывает заготовку будущего меча. Затем старший ученик мастера закаливает меч в трех водах – кипящей, студеной и теплой. И в конце мастер собственноручно изготавливает рукоять меча и наносит узоры. Создать многопоточное приложение, в котором мастер и его ученики представлены разными потоками.

7. а) Определить количество чисел m , являющихся квадратами некоторого целого числа, в матрице A . Заменить все простые числа в A на m . Входные данные: целое положительное число n , массив чисел A размерности $n \times n$.

б) Дана последовательность натуральных чисел $a_0 \dots a_{n-1}$. Создать многопоточное приложение для поиска суммы $\sum a_i$, где a_i – четные числа.

8. а) Найти сумму индексов всех седловых точек матрицы A . Будем считать элемент седловой точкой, если он является наименьшим в своей строке и наибольшим в своем столбце, либо наоборот. Входные данные: целое положительное число n , массив чисел A размерности $n \times n$.

б) Дана последовательность натуральных чисел $a_0 \dots a_{n-1}$. Создать многопоточное приложение для вычисления выражения $a_0 - a_1 + a_2 - a_3 + a_4 - a_5 + \dots$.

9. а) Найти максимальный и минимальный среди тех элементов матрицы A , сумма индексов которых равна двойке в любой целочисленной степени. Входные данные: целое положительное число n , целое положительное число k , массив чисел от A размерности $n \times k$.

б) Дана последовательность натуральных чисел $a_0 \dots a_{n-1}$. Создать многопоточное приложение для поиска всех a_i , являющихся квадратами любого натурального числа.

10. а) Определить, является ли матрица A симметричной относительно главной диагонали. Входные данные: целое положительное число n , массив чисел A размерности $n \times n$. Использовать не менее 4 процессов для решения задачи.

б) Дана последовательность натуральных чисел $a_0 \dots a_{n-1}$. Создать многопоточное приложение для поиска всех a_i , являющихся простыми числами.

11. а) В матрице A найти в каждой строке наибольший элемент и поменять его местами с элементом, стоящим на главной диагонали и в той же строке. Входные данные: целое положительное число n , массив чисел A размерности $n \times n$. Использовать n или $n+1$ процессов для решения задачи.

б) Дана последовательность натуральных чисел $a_0 \dots a_{n-1}$. Создать многопоточное приложение для поиска минимального a_i .

12. а) Упорядочить по возрастанию элементы в каждой строке матрицы A . Входные данные: целое положительное число n , целое положительное число k , массив чисел от A размерности $n \times k$. Использовать n или $n+1$ процессов для решения задачи.

б) Дана последовательность натуральных чисел $a_0 \dots a_{n-1}$. Создать многопоточное приложение для поиска произведения чисел $a_0 * a_1 * \dots * a_{n-1}$.

13. а) В массиве строк хранятся фамилии и оценки учащихся. Найти учащегося с максимальным средним баллом, вывести список всех учащихся, имеющих однофамильцев. Входные данные: целое положительное число n , массив строк размерности n . Использовать n или $n+1$ процессов для решения задачи.

б) Дана последовательность символов $C = \{c_0 \dots c_{n-1}\}$ и символ b . Создать многопоточное приложение для определения количества вхождений символа b в строку C .

14. а) Найти произведение всех элементов в матрице A , сумма или разность индексов которых является простым числом, отрицательные разности не рассматривать. Входные данные: целое положительное число n , целое положительное число k , массив чисел от A размерности $n \times k$.

б) Дана последовательность символов $C = \{c_0 \dots c_{n-1}\}$. Дан набор из N пар кодирующих символов (a_i, b_i) . Создать многопоточное приложение, кодирующее строку C следующим образом: строка разделяется на подстроки и каждый поток осуществляет кодирование своей подстроки.

15. а) Вычислить произведение матрицы A на B , где матрица B получена из матрицы A , заменой отрицательных элементов нулем и последующим транспонированием. Входные данные: целое положительное число n , массив чисел A размерности $n \times n$.

б) Дана последовательность натуральных чисел $a_0 \dots a_{n-1}$. Создать многопоточное приложение для поиска суммы квадратов a_i^2 .

16. а) Двоичные числа записаны в строке, разделителем является пробел. Количество чисел равно m . Найти сумму всех двоичных чисел как двоичное число и как десятичное число. Входные данные: строка S . Для решения задачи использовать не менее m процессов.

б) Дана последовательность символов $C = \{c_0 \dots c_{n-1}\}$. Дан набор из N пар кодирующих символов (a_i, b_i) . Создать многопоточное приложение, кодирующее строку C следующим образом: поток 0 заменяет в строке C все символы a_0 на символы b_0 , поток 1 заменяет в строке C все символы a_1 на символы b_1 , и т.д. Потоки должны осуществлять кодирование последовательно.

17. а) Проверить, можно ли составить слово S из элементов символьного массива C . Учитывать количество требуемых символов для составления слова. Входные данные: строка S , массив символов C . Использовать для решения задачи столько процессов, сколько неповторяющихся символов в строке S .

б) Даны последовательности символов $A = \{a_0 \dots a_{n-1}\}$ и $C = \{c_0 \dots c_{k-1}\}$. В общем случае $n \neq k$. Создать многопоточное приложение, определяющее, совпадают ли посимвольно строки A и C .

18. а) Строка содержит произвольный русский текст. Вывести, сколько раз в ней встречается буква а, буква б, буква в и т.д. Найти три наиболее часто встречающиеся буквы. Входные данные: строка S .

Использовать для решения задачи столько процессов, сколько букв в русском алфавите.

б) Дана квадратная матрица A . Создать многопоточное приложение для поиска сумм строк и столбцов.

19. а) Найти максимальный по модулю элемент в матрице A и его индексы – x, y . Поменять местами строку x со строкой k и столбец y со столбцом k . Входные данные: целое положительное число n , массив чисел A размерности $n \times n$, целое положительное число $k \neq 0$ и $\neq n-1$.

б) Дана последовательность натуральных чисел $\{a_0 \dots a_{n-1}\}$. Создать многопоточное приложение для поиска максимального a_i .

20. а) В массиве A заменить отрицательные элементы нулями, а положительные элементы единицами. Перевести полученное двоичное число в десятичную систему счисления. Входные данные: целое положительное число n , массив чисел A размерности n , целое положительное число $k \neq 2$ и $\neq n/2$. Использовать для решения задачи k процессов.

б) Изготовление знаменитого самурайского меча катаны происходит в три этапа. Младший ученик мастера выковывает заготовку будущего меча. Затем старший ученик мастера закаливает меч в трех водах – кипящей, студеной и теплой. И в конце мастер собственноручно изготавливает рукоять меча и наносит узоры. Требуется создать многопоточное приложение, в котором мастер и его ученики представлены одинаковыми потоками (обработка производится в цикле).

Контрольные вопросы

1. Какие механизмы относятся к средствам межпроцессной коммуникации? Какие у них задачи?
2. Понятие о System V IPC, что это? Какие их свойства?
3. Что мы называем пространством имен?
4. Опишите функцию `ftok()`.
5. Дескрипторы System V IPC. Разделяемая память в UNIX.
6. Системные вызовы `shmget()`, `shmat()`, `shmdt()`.
7. Зачем потоку добровольно отказываться от центрального процессора, вызывая процедуру `threadyield()`? Ведь в отсутствие периодических таймерных прерываний он может вообще уже никогда не вернуть себе центральный процессор.

8. Может ли поток быть приостановлен таймерным прерыванием? Если да, то при каких обстоятельствах, а если нет, то почему?
9. В чем заключается самое большое преимущество от реализации потоков в пользовательском пространстве? А в чем заключается самый серьезный недостаток?
10. Рассмотрим систему, в которой потоки реализованы целиком в пользовательском пространстве, а система поддержки исполнения программ раз в секунду получает таймерное прерывание.
11. Предположим, таймерное прерывание происходит в тот самый момент, когда какой-то поток выполняется в системе поддержки исполнения программ. К возникновению какой проблемы это может привести?
12. Можете ли вы предложить способ разрешения этой проблемы?
13. Какие стеки существуют в системе, имеющей потоки, реализованные на пользовательском уровне: по одному стеку на поток или по одному стеку на процесс? Ответьте на тот же вопрос при условии, что потоки реализованы на уровне ядра. Обоснуйте ответ.
14. Обычно в процессе разработки компьютера его работа сначала моделируется программой, которая запускает строго по одной команде. Таким сугубо последовательным способом моделируются даже многопроцессорные компьютеры. Может ли при отсутствии одновременных событий, как в данном случае, возникнуть составная ситуация?

Лабораторная работа 6

СЕМАФОРЫ В UNIX КАК СРЕДСТВО СИНХРОНИЗАЦИИ ПРОЦЕССОВ

Цели и задачи

Ознакомиться с общими принципами работы семафоров. Научиться использовать семафоры для синхронизации процессов и потоков и для защиты критических секций.

6.1 Семафоры в UNIX.

Отличие операций над UNIX-семафорами от классических операций

Одним из первых механизмов, предложенных для синхронизации поведения процессов, стали семафоры, концепцию которых описал Дейкстра (Dijkstra) в 1965 году. При разработке средств System V IPC семафоры вошли в их состав как неотъемлемая часть. Общими ресурсами процессов являются файлы, сегменты разделяемой памяти. Возможность одновременного изменения несколькими процессами общих данных называют критической секцией, так как такая совместная работа процессов может привести к возникновению ошибок. Например, если несколько процессов осуществляют запись данных в один и тот же файл, эти данные могут оказаться перемешанными. Наиболее простой механизм защиты критической секции состоит в расстановке «замков», пропускающих только один процесс для выполнения критической секции и останавливающий все остальные процессы, пытающиеся выполнить критическую секцию, до тех пор, пока эту критическую секцию не выполнит пропущенный процесс. Семафоры позволяют выполнять такую операцию, как и многие другие.

Использование общих данных несколькими процессами может привести к ошибкам и конфликтам. Но при этом семафоры и сами являются общими данными. Такое положение не является противоречивым, в силу того, что:

- 1) значение семафора расположено не в адресном пространстве некоторого процесса, а в адресном пространстве ядра;

2) операция проверки и изменения значения семафора, вызываемая процессом, является атомарной, т. е. не прерываемой другими процессами. Эта операция выполняется в режиме ядра.

Общими данными процессов также являются каналы и сообщения. Но операции с каналами и сообщениями защищаются системой, и, как правило, программист не должен использовать семафор для защиты записи сообщения в очередь сообщений либо записи данных в канал. Однако это не всегда правильно. Например, система обеспечивает атомарную запись в канал только для данных не больше определенного объема.

Набор операций над семафорами **System V IPC** отличается от классического набора операций $\{P, V\}$, предложенного Дейкстры. Он включает три операции:

$A(S, n)$ – увеличить значение семафора S на величину n ;

$D(S, n)$ – пока значение семафора $S < n$, процесс блокируется. Далее $S = S - n$;

$Z(S)$ – процесс блокируется до тех пор, пока значение семафора S не станет равным 0.

Изначально все IPC-семафоры иницируются нулевым значением.

Классической операции $P(S)$ соответствует операция $D(S,1)$, а классической операции $V(S)$ соответствует операция $A(S,1)$. Аналогом ненулевой инициализации семафоров Дейкстры значением n может служить выполнение операции $A(S,n)$ сразу после создания семафора S , с обеспечением атомарности создания семафора и ее выполнения посредством другого семафора. Классические семафоры реализуются через семафоры **System V IPC**. Обратное не является верным. Используя операции $P(S)$ и $V(S)$, не получится реализовать операцию $Z(S)$.

Поскольку IPC-семафоры являются составной частью средств **System V IPC**, то для них верно все, что говорилось об этих средствах в материалах предыдущего семинара. IPC-семафоры являются средством связи с непрямой адресацией, требуют инициализации для организации взаимодействия процессов и специальных действий для освобождения системных ресурсов по его окончании. Пространством имен IPC-семафоров является множество значений ключа, генерируемых с помощью функции **ftok()**. Для совершения операций

над семафорами системным вызовом в качестве параметра передаются **ИПС**-дескрипторы семафоров, однозначно идентифицирующих их во всей вычислительной системе, а вся информация о семафорах располагается в адресном пространстве ядра операционной системы. Это позволяет организовывать через семафоры взаимодействие процессов, даже не находящихся в системе одновременно.

6.2 Создание массива семафоров или доступ к уже существующему. Системный вызов `semget()`

В целях экономии системных ресурсов операционная система **UNIX** позволяет создавать не по одному семафору для каждого конкретного значения ключа, а связывать с ключом целый массив семафоров (в Linux – до 500 семафоров в массиве, хотя это количество может быть уменьшено системным администратором). Для создания массива семафоров, ассоциированного с определенным ключом, или для доступа по ключу к уже существующему массиву используется системный вызов `semget()`, являющийся аналогом системного вызова `shmget()` для разделяемой памяти, который возвращает значение **ИПС**-дескриптора для этого массива. При этом применяются те же способы создания и доступа, что и для разделяемой памяти. Вновь созданные семафоры инициализируются нулевым значением.

Системный вызов `semget()`

Прототип системного вызова

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

```
#include <sys/ipc.h>
```

```
#include <sys/sem.h>
```

```
int semget(key_t key, int nsems,  
           int semflg);
```

Описание системного вызова

Системный вызов `semget` предназначен для выполнения операции доступа к массиву **ИПС**-семафоров и, в случае ее успешного завершения, возвращает дескриптор **System V IPC** для этого массива (целое неотрицательное число, однозначно характеризующее массив семафоров внутри вычислительной системы и использующееся в дальнейшем для других операций с ним).

Параметр **key** является ключом **System V IPC** для массива семафоров, т. е. фактически его именем из пространства имен **System V IPC**. В качестве значения этого параметра может использоваться значение ключа, полученное с помощью функции **ftok()**, или специальное значение **IPC_PRIVATE**. Использование значения **IPC_PRIVATE** всегда приводит к попытке создания нового массива семафоров с ключом, который не совпадает со значением ключа ни одного из уже существующих массивов и не может быть получен с помощью функции **ftok()** ни при одной комбинации ее параметров.

Параметр **nsems** определяет количество семафоров в создаваемом или уже существующем массиве. В случае если массив с указанным ключом уже имеется, но его размер не совпадает с указанным в параметре **nsems**, констатируется возникновение ошибки.

Параметр **semflg** – флаги – играет роль только при создании нового массива семафоров и определяет права различных пользователей при доступе к массиву, а также необходимость создания нового массива и поведение системного вызова при попытке создания. Он является некоторой комбинацией (с помощью операции побитовое или – "**|**") следующих предопределенных значений и восьмеричных прав доступа:

IPC_CREAT – если массива для указанного ключа не существует, он должен быть создан;

IPC_EXCL – применяется совместно с флагом **IPC_CREAT**. При совместном их использовании и существовании массива с указанным ключом доступ к массиву не производится и констатируется ошибка, при этом переменная **errno**, описанная в файле **<errno.h>**, примет значение **EEXIST**;

0400 – разрешено чтение для пользователя, создавшего массив;

0200 – разрешена запись для пользователя, создавшего массив;

0040 – разрешено чтение для группы пользователя, создавшего массив;

0020 — разрешена запись для группы пользователя, создавшего массив;

0004 — разрешено чтение для всех остальных пользователей;

0002 — разрешена запись для всех остальных пользователей.

Вновь созданные семафоры иницируются нулевым значением.

Возвращаемое значение

Системный вызов возвращает значение дескриптора **System V IPC** для массива семафоров при нормальном завершении и значение **-1** при возникновении ошибки.

6.3 Семафоры для синхронизации процессов

Семафор обладает внутренним значением – целым числом, принадлежащим типу **unsigned short**. Семафоры могут быть объединены в единую группу.

int semop(int semid, sembuf*semop, size_t nops)

Функция выполняет над группой семафоров с дескриптором **semid** набор операций **semop**, **nops** – количество операций, выполняемых из набора **semop**.

Для задания операции над группой семафоров используется структура **sembuf**.

Первый параметр структуры **sembuf** определяет порядковый номер семафора в группе. Семафоры в группе индексируются с нуля.

Второй параметр структуры **sembuf** представляет собой целое число = **S** и определяет действие, которое необходимо произвести над семафором, с индексом, записанным в первом параметре.

Если **S>0**, к внутреннему значению семафора добавляется число **S**. Эта операция не блокирует процесс.

Если **S=0**, процесс приостанавливается, пока внутреннее значение семафора не станет равно нулю.

Если **S<0**, процесс должен отнять от внутреннего значения семафора модуль **S**.

Если значение семафора - $|S| \geq 0$, производится вычитание и процесс продолжает свою работу. Если значение семафора - $|S| < 0$, процесс останавливается до тех пор, пока другой процесс не увеличит значение семафора на достаточную величину, чтобы операция вычитания выдала неотрицательный результат. Тогда производится операция вычитания и процесс продолжает свою работу. Например, если значение семафора равно трем, а процесс пытается выполнить над ним операцию -4, этот процесс будет заблокированным, пока значение семафора не увеличится хотя бы на единицу.

Третий параметр структуры **sembuf** может быть равен **0**, тогда операция **S** \square **0** будет предполагать блокировку процесса, т. е. выполняться так, как описано выше. Также **sembuf** может быть равен **IPC_NOWAIT**, в этом случае работа процесса не будет останавливаться. Если процесс будет пытаться выполнить вычитание от значения семафора, дающее отрицательный результат, эта операция просто игнорируется и процесс продолжает выполнение.

Последний параметр в функции **semop** определяет количество операций, берущихся для выполнения из второго параметра функции.

Т. е. следующий вызов

```
sembuf Minus4 = {0, -4, 0} ;
```

```
semop( semid, &Minus4, 1) ; нельзя заменить таким вызовом.
```

```
sembuf Minus1 = {0, -1, 0} ;
```

```
semop( semid, &Minus1, 4) ; корректной заменой может являться.
```

```
sembuf Minus1[4] = {0, -1, 0, 0, -1, 0, 0, -1, 0, 0, -1, 0} ;
```

```
semop( semid, Minus1, 4) ;
```

Для иллюстрации сказанного рассмотрим простейшие программы, синхронизирующие свои действия с помощью семафоров

Программа 6-01a для иллюстрации работы с семафорами

Эта программа получает доступ к одному системному семафору, ждет, пока его значение не станет больше или равным 1 после запусков программы 05-1b, затем уменьшает его на 1

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

```
#include <sys/ipc.h>
```

```
#include <sys/sem.h>
```

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

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    int semid; /* IPC дескриптор для массива IPC семафоров */
```

```
    char pathname[] = "Имя файла"; /* Имя файла,
```

```
    использующееся для генерации ключа. Файл с таким именем  
    должен существовать в текущей директории */
```

```

key_t key; /* IPC ключ */
struct sembuf mybuf; /* Структура для задания операции над
семафором */
/* Генерирование IPC-ключа из файла в текущей директории и
номера экземпляра массива семафоров 0 */

```

```

if((key = ftok(pathname,0)) < 0)
{
    cout<<"Can't generate key"<<endl;
}

```

/ Попытка получить доступ по ключу к массиву семафоров, если он существует, или создать его из одного семафора, если его еще не существует, с правами доступа read & write для всех пользователей */*

```

if((semid = semget(key, 1, 0666 | IPC_CREAT)) < 0)
{
    cout<<"Can't get semid"<<endl;
}

```

/ Выполнение операции D(semid1,1) для массива семафоров. Для этого сначала необходимо заполнить структуру. Флаг, равный 0. Массив семафоров состоит из одного семафора с номером 0. Код операции -1.*/*

```

mybuf.sem_op = -1;
mybuf.sem_flg = 0;
mybuf.sem_num = 0;
if(semop(semid, &mybuf, 1) < 0)
{
    cout<<"Can't wait for condition"<<endl;
}
cout<<"Condition is present"<<endl;
return 0;
}

```

Листинг 6-01а. Программа 6-01а для иллюстрации работы с семафорами.

Программа 6-01b для иллюстрации работы с семафорами

```
/* Эта программа получает доступ к одному системному семафору и
увеличивает его на 1 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
#include <iostream>
using namespace std;
int main()
{
    int semid; /* IPC дескриптор для массива IPC семафоров */
    char pathname[] = "Имя файла"; /* Имя файла,
используемое для генерации ключа. Файл с таким именем
должен существовать в текущей директории */
    key_t key; /* IPC ключ */
    struct sembuf mybuf; /* Структура для задания операции над
семафором */
    /* Генерирование IPC ключа из имени файла 08-1a.c в текущей
директории и номера экземпляра массива семафоров 0 */
    if((key = ftok(pathname,0)) < 0)
    {
        cout<<"Can't generate key"<<endl;
    }
    /* Попытка получить доступ по ключу к массиву семафоров,
если он существует, или создать его из одного семафора, если его еще
не существует, с правами доступа read & write для всех
пользователей */
    if((semid = semget(key, 1, 0666 | IPC_CREAT)) < 0)
    {
        cout<<"Can't get semid"<<endl;
    }
    /* Выполнение операции A(semid,1) для массива семафоров.
Для этого сначала необходимо заполнить структуру. Флаг, равный 0.
Массив семафоров состоит из одного семафора с номером 0. Код
операции 1. */
    mybuf.sem_op = 1;
```

```

mybuf.sem_flg = 0;
mybuf.sem_num = 0;
if(semop(semid, &mybuf, 1) < 0)
{
    cout<<"Can't wait for condition"<<endl;
}
cout<<"Condition is set "<<endl;
return 0;
}

```

Листинг 6-01b. Программа 6-01b для иллюстрации работы с семафорами

Первая программа выполняет над семафором S операцию $D(S,1)$, вторая программа выполняет над тем же семафором операцию $A(S,1)$. Если семафора в системе не существует, любая программа создает его перед выполнением операции. Поскольку при создании семафор всегда инициализируется 0, то программа 1 может работать без блокировки только после запуска программы 2.

Программа 6-02b, пример использования семафора для синхронизации процессов

Процесс-родитель создает четыре процесса-потомка и ожидает их завершения, используя для этого семафор.

```

#include <stdio.h>
#include <unistd.h>
#include <sys/sem.h>
#include <sys/ipc.h>
#include <iostream>
using namespace std;
int main()
{
    int semid ; //для хранения дескриптора группы семафоров
                //создать группу семафоров, состоящую из одного семафора
                semid = semget (IPC_PRIVATE, 1, IPC_CREAT|0666) ;
                if ( semid < 0 )
                //если не удалось создать группу семафоров, завершить выполнение

```

```

{
cout<< "Ошибка"<<endl;
return 0 ;
}

sembuf Plus1 = {0, 1, 0} ;
//операция прибавляет единицу к семафору с индексом 0
sembuf Minus4 = {0, -4, 0} ;
//операция вычитает 4 от семафора с индексом 0
//создать четыре процесса-потомка
for (int i=0 ; i<4 ; i++)
{
if ( fork() == 0 ) //истинно для дочернего процесса
{
cout<<"hi"<< i << "!"<<endl;
//здесь должен быть код, выполняемый процессом-потомком
//добавить к семафору единицу, по окончании работы
semop( semid, &Plus1, 1) ;
return 1 ;
}
}
semop( semid, &Minus4, 1) ;
return 1 ;
}

```

Листинг 6-02b. Пример использования семафора для синхронизации процессов

В описанном примере новый семафор при создании обладает нулевым значением. Каждый из четырех порожденных процессов после выполнения своих вычислений увеличивает значение семафора на единицу. Родительский процесс пытается уменьшить значение семафора на четыре. Таким образом, процесс-родитель останется заблокированным до тех пор, пока не отработают все его потомки.

6.4 Использование семафора для защиты критической секции

Использование семафора для синхронизации доступа нескольких процессов к общему ресурсу, т. е. для защиты критической секции. Общим ресурсом будет являться разделяемая память.

Процесс-родитель создает сегмент разделяемой памяти и порождает три дочерних процесса, процесс-родитель и его потомки вычисляют сумму элементов определенной части массива и записывают вычисленную сумму в разделяемую память. Родительский процесс дожидается окончания работы потомков и выводит окончательный результат – сумму всех элементов массива.

Так как четыре процесса могут изменять данные разделяемой памяти, необходимо сделать это изменение атомарным для каждого процесса. Для этого необходимо создать семафор, который будет принимать два значения – ноль и единицу.

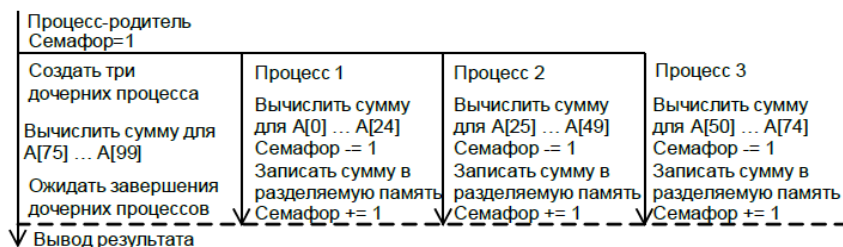


Рис. 6.1. Схема процессов

Программа 6-03а. Использование семафора для синхронизации доступа нескольких процессов к общему ресурсу (разделяемой памяти)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/wait.h>
#include <iostream>
using namespace std;
int shmid ; //для хранения дескриптора разделяемой памяти
int semid ; //для хранения дескриптора группы семафоров
sembuf Plus1 = {0, 1, 0} ; //операция прибавляет единицу к семафору с индексом 0
sembuf Minus1 = {0, -1, 0} ; //операция вычитает единицу от семафора с индексом 0
```

```

int A[100] ; //массив, сумма элементов которого вычисляется
процессами
struct тутет //структура, под которую будет выделена разделяемая
память
{
    int sum ; //для записи суммы
} *mem_sum;
void summa (int p) //для вычисления суммы части элементов массива
{
    int i, sum = 0 ; //для суммирования элементов
    int begin = 25*p ; //индекс массива, с которого начинается
суммирование
    int end = begin+25 ; //индекс массива, на котором завершается
суммирование
    for(i=begin; i<end; i++) sum+=A[i]; //вычисление суммы части
элементов массива
    semop( semid, &Minus1, 1) ; //отнять единицу от семафора
    mem_sum->sum+=sum; //добавление вычисленного результата в
общую переменную
    semop( semid, &Plus1, 1); //добавить единицу к семафору
}
int main()
{ //запрос на создание разделяемой памяти объемом 2 байта
shmid = shmget(IPC_PRIVATE, 2, IPC_CREAT|0666);
//если запрос оказался неудачным, завершить выполнение
if (shmid < 0 )
{
    cout<< "Ошибка"<<endl;
    return 0;
}
//создать группу семафоров, состоящую из одного семафора
semid = semget (IPC_PRIVATE, 1, IPC_CREAT|0666);
//если не удалось создать группу семафоров, завершить выполнение
if ( semid < 0 )
{
    cout<< "Ошибка"<<endl;
    return 0;
}
}

```

```

semop( semid, &Plus1, 1) ; //теперь семафор равен единице
//теперь тем_sum указывает на выделенную разделяемую память
tem_sum = (тутем *)shmat(shmid,NULL,0) ;
tem_sum->sum = 0;
//тут должна быть инициализация элементов массива A
//создать три процесса-потомка
for (int i=0; i<3; i++)
{
if (fork() == 0 ) //истинно для дочернего процесса
{
    summa(i) ;
    return 1 ;
}
}
summa(3); //родительский процесс вычисляет последнюю четверть
массива
for (int i=0; i<3; i++)
wait(NULL) ; //дождаться завершения процессов-потомков
//вывести на экран сумму всех элементов массива
cout<< "Результат = " << tem_sum->sum << endl;
return 1;
}

```

Листинг 6-03. Пример использования семафора для защиты критической секции

Программа 6-04. Иллюстрация использования семафора для защиты критической секции

```

#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/wait.h>
#include <iostream>
using namespace std;
int shmid;
int semid;

```



```

sembuf Plus1 = {0, 1, 0};
sembuf Minus1 = {0, -1, 0};
int A[100];
struct mymem
{ int sum;
  } *mem_sum;
void summa (int p)
{
  inti, sum = 0;
  int begin =25*p;
  int end = begin+25;
  for(i=begin ; i<end ; i++) sum+=A[i];
  semop( semid, &Minus1, 1);
  mem_sum->sum+=sum;
  semop( semid, &Plus1, 1);
}
int main()
{
  shmid = shmget(IPC_PRIVATE, 2, IPC_CREAT/0666);
  if (shmid< 0 )
  {
    cout<<"Ошибка"<< endl;
    return 0;
  }
  semid = semget (IPC_PRIVATE, 1, IPC_CREAT/0666);
  if ( semid< 0 )
  {
    cout<<"Ошибка"<< endl;
    return 0;
  }
  semop( semid, &Plus1, 1);
  mem_sum = (mymem *)shmat(shmid,NULL,0);
  mem_sum->sum = 0;
  A[10]=5;
  A[30]=5;
  A[60]=5;
  A[90]=5;
  for (inti=0 ; i<3 ; i++)
  {

```

```

if ( fork() == 0 )
{
    summa(i);
    return 1;
}
}
summa(3);
for (inti=0 ; i<3 ; i++)
wait(NULL);
cout<<"Calculate= "<<mem_sum->sum<< endl;
return 1;
}

```

Листинг 6-04. Программа, иллюстрирующая использование семафора для защиты критической секции

Семафор получает при создании значение, равное нулю, которое сразу же устанавливается в единицу. Первый процесс, вызвавший функцию `semop(semid, &Minus1, 1)`, уменьшает значение семафора до нуля, переходит к записи в разделяемую память и по завершении операции записи устанавливает значение семафора в единицу, вызвав `semop(semid, &Plus1, 1)`. Если управление перейдет к другому процессу во время записи в разделяемую память первым процессом, вызов функции «отнять от семафора единицу» остановит работу другого процесса, до того момента, когда значение семафора не станет положительным. Что может произойти только тогда, когда первый процесс завершит запись в общую память и выполнит операцию «добавить к семафору единицу».

Если процессы обладают несколькими общими ресурсами, то необходимо для каждого общего ресурса создавать свой семафор.

6.5 Семафоры для синхронизации потоков

Для многопоточного приложения, как и для многопроцессного, критической секцией является изменение несколькими потоками общего ресурса, например файла или глобальной переменной. Для синхронизации работы потоков и для синхронизации доступа нескольких потоков к общим ресурсам используются семафоры. Но при этом семафоры, используемые для синхронизации потоков,

принадлежат к другому стандарту, чем семафоры, используемые для синхронизации процессов.

Каждый семафор содержит неотрицательное целое значение. Любой поток может изменять значение семафора. Когда поток пытается уменьшить значение семафора, происходит следующее: если значение больше нуля, то оно уменьшается, если же значение равно нулю, поток приостанавливается до того момента, когда значение семафора станет положительным, тогда значение уменьшается и поток продолжает работу. Увеличение значения семафора возможно всегда, эта операция не предполагает блокировки. Однако значение семафора не должно выходить за границы типа **unsigned int**.

Функции для работы с семафорами (заголовочный файл `semaphore.h`):

int sem_init (sem_t *sem, int pshared, unsigned int value)

Функция инициализирует семафор **sem** и присваивает ему значение **value**. Если параметр **pshared** больше нуля, семафор может быть доступен нескольким процессам, если **pshared** равен нулю, семафор создается для использования внутри одного процесса.

Функция возвращает ноль в случае успеха.

Int sem_destroy (sem_t *sem)

Функция разрушает семафор **sem** и возвращает ноль в случае успеха.

int sem_getvalue (sem_t *sem, int *sval)

Функция записывает значение семафора **sem** в ***sval**. Если несколько потоков используют семафор, полученное значение семафора может быть устаревшим.

int sem_post (sem_t *sem)

Функция увеличивает значение семафора **sem** на единицу.

int sem_wait (sem_t *sem)

Если текущее значение семафора больше нуля, функция уменьшает значение семафора **sem** на единицу. Если текущее значение семафора равно нулю, выполнение потока, вызвавшего функцию **sem_wait**, приостанавливается до тех пор, когда значение семафора станет положительным, тогда значение семафора уменьшается на единицу и поток продолжает работу.

int sem_trywait (sem_t *sem)

Если текущее значение семафора больше нуля, функция уменьшает значение семафора **sem** на единицу и возвращает ноль. Если текущее значение семафора равно нулю, функция возвращает не нулевое значение. Функция **sem_wait** не останавливает работу потока.

Обратите внимание!

Для корректной работы описанных семафоров, необходимо при компиляции программы использовать команду:

g++ <исходный файл> -o <исполняемый файл> -lpthread.

Программа 6-05а. Пример использования семафора для синхронизации потоков.

Перепишем пример использования потоков из предыдущей работы, введя семафор для защиты глобальной переменной SUM и семафор, блокирующий процесс-родитель, пока не выполнятся все его дочерние потоки. Изменения выделены жирным шрифтом.

```
#include <sched.h>
#include <stdio.h>
#include <unistd.h>
#include <semaphore.h>
#include <iostream>
using namespace std;
#define NUMSTACK 5000 //объем стека для отдельного потока
sem_t sem ; //семафор для защиты критической секции
sem_t sem4 ; //семафор для синхронизации родителя и потомков
int A[100] ; //массив, сумма элементов которого вычисляется
потоками
int SUM=0; //для записи общей суммы
char stack[4][ NUMSTACK]; ////для хранения стека четырех потоков
int func(void *param) //стартовая функция потоков
{
    int i, sum = 0; //для суммирования элементов
    //индекс массива, с которого начинается суммирование
    int p =(int *)param;
    p=p*25;
```

```

for(i=p; i<p+25; i++) sum+=A[i] ; //вычисление суммы части
элементов массива
sem_wait(&sem); //отнять единицу от значения семафора sem
SUM+=sum; //добавление вычисленного результата в общую
переменную
sem_post(&sem); //добавить единицу к значения семафора sem
sem_post(&sem4); //добавить единицу к значения семафора sem4
return 1;
}
int main()

```

//тут должна быть инициализация элементов массива A

```

sem_init (&sem, 1, 1); //инициализация семафора sem со значением 1
sem_init (&sem4, 1, 0); //инициализация семафора sem4 со значением 0
int param[4]; //для хранения параметров потоков
for (int i=0; i<4; i++) //создание четырех потоков
{
param[i]=i;
char *tostack=stack[i];
clone(func,(void*)( tostack+ NUMSTACK -1), CLONE_VM, (void *)
(param+i));
}
//отнять четыре единицы от значения семафора sem4
for (71nti =0; i<4; i++) sem_wait(&sem4);
cout<< "Результат = " <<SUM <<endl ;
return 1;
}

```

Листинг 6-05. Пример использования семафора для синхронизации потоков

Принцип работы семафоров, использованных в данном примере, полностью аналогичен семафорам из двух предыдущих примеров – пример использования семафора для синхронизации процессов и пример использования семафора для защиты критической секции.

6.6 Удаление набора семафоров из системы с помощью команды **ipcrm** или системного вызова **semctl()**

Массив семафоров может продолжать существовать в системе и после завершения использовавших его процессов, а семафоры будут сохранять свое значение. Это может привести к некорректному поведению программ, предполагающих, что семафоры были только что созданы и, следовательно, имеют нулевое значение. Необходимо удалять семафоры из системы перед запуском таких программ или перед их завершением. Для удаления семафоров можно воспользоваться командами **ipcs** и **ipcrm**, рассмотренными в материалах предыдущего семинара. Команда **ipcrm** в этом случае должна иметь вид

ipcrm sem <IPC идентификатор>

Для этой же цели можно применять системный вызов **semctl()**.

Системный вызов **semctl()**

Прототип системного вызова

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, union semun arg);

Описание системного вызова

Системный вызов **semctl** предназначен для получения информации о массиве **IPC**-семафоров, изменения его атрибутов и удаления его из системы. Данное описание не является полным описанием системного вызова. Для изучения полного описания обращайтесь к **UNIX Manual**. В данном курсе применяется системный вызов **semctl** только для удаления массива семафоров из системы. Параметр **semid** является дескриптором **System V IPC** для массива семафоров, т. е. значением, которое вернул системный вызов **semget()** при создании массива или при его поиске по ключу.

В качестве параметра **cmd** в рамках нашего курса мы всегда будем передавать значение **IPC_RMID** – команду для удаления сегмента

разделяемой памяти с заданным идентификатором. Параметры **semnum** и **arg** для этой команды не используются, поэтому мы всегда будем подставлять вместо них значение **0**.

Если какие-либо процессы находились в состоянии «ожидание» для семафоров из удаляемого массива при выполнении системного вызова **semop()**, то они будут разблокированы и вернутся из вызова **semop()** с индикацией ошибки.

Возвращаемое значение

Системный вызов возвращает значение **0** при нормальном завершении и значение **-1** при возникновении ошибки.

6.7 Понятие о POSIX-семафорах

В стандарте **POSIX** вводятся другие семафоры, полностью аналогичные семафорам Дейкстры. Для инициализации значения таких семафоров применяется функция **sem_init()**, аналогом операции **P** служит функция **sem_wait()**, а аналогом операции **V** – функция **sem_post()**. К сожалению, в **Linux** такие семафоры реализованы только для нитей исполнения одного процесса.

Порядок выполнения лабораторной работы

1. Наберите программы, сохраните под именами **6-01a.** и **6-01b.** соответственно, откомпилируйте и проверьте правильность их поведения.
2. Измените программы **6-01a.** и **6-01b.** так, чтобы первая программа могла работать без блокировки после не менее 5 запусков второй программы.
3. Посмотреть в динамике работу семафоров для созданных приложений, используя команду **ipcs -s**.
4. В задании а) лабораторной работы 5 ввести защиту критических секций с помощью семафора. Если у вашего алгоритма отсутствуют критические секции, объяснить, почему их нет.
5. В задании б) лабораторной работы 5 ввести защиту критических секций и обеспечить синхронизацию между процессом-родителем и дочерними потоками с помощью семафоров. Если у вашего алгоритма

отсутствуют критические секции, объяснить, почему их нет.

6. Модифицируйте программы из раздела «Необходимость синхронизации процессов и нитей исполнения, использующих общую память» лабораторной работы 5, которые иллюстрировали некорректную работу через разделяемую память, обеспечив с помощью семафоров взаимного исключения для их правильной работы.

7. Организуйте двустороннюю поочередную связь процесса-родителя и процесса-ребенка через `pipe`, используя для синхронизации семафоры, модифицировав программу из раздела "Организация связи через `pipe` между процессом-родителем и процессом потомком» лабораторной работы 4.

8. Разработать алгоритм решения индивидуального задания лабораторной работы 6 с учетом разделения вычислений между несколькими процессами. Для обмена информацией между процессами использовать разделяемую память. Для защиты операций с разделяемой памятью и синхронизации процессов использовать семафоры. Реализовать алгоритм и протестировать его на нескольких примерах.

9. Разработать алгоритм решения индивидуального задания лабораторной работы 6, с учетом разделения вычислений между несколькими потоками. Для синхронизации потоков и защиты критических секций использовать семафоры. Реализовать алгоритм и протестировать его на нескольких примерах.

Варианты индивидуальных заданий

1. Найти максимальный элемент из минимальных элементов в каждой строке матрицы A . Входные данные: целое положительное число n , целое положительное число k , массив чисел от A размерности $n \times k$. Использовать n или $n+1$ процессов (потоков) для решения задачи.

2. Найти наибольший элемент на главной диагонали и наименьший элемент на побочной диагонали, заменить элемент, стоящий на пересечении диагоналей, на сумму двух найденных значений. Входные данные: целое положительное нечетное число n , массив чисел от A размерности $n \times n$.

3. Определить все целые числа из интервала $[A, B]$, имеющие наибольшее количество делителей. Найти среднее арифметическое найденных чисел. Входные данные: число A , число B , целое

положительное число $k > 1$ и $< (B-A)/2$. Использовать k процессов (потоков) для решения задачи.

4. Найти все простые натуральные числа из интервала $[A, B]$, двоичная запись которых является палиндромом – одинаково читается как слева направо, так и справа налево. Вычислить сумму найденных чисел. Входные данные: число A , число B .

5. Задана строка S , содержащая не менее двух слов, и символ c . Составить новую строку $S1$ из слов строки S , в которых есть символ c , и новую строку $S2$ из слов строки S , в которых нет символа c . Учитывать порядок вхождения слов в строку. Входные данные: строка S произвольной длины и символ c . Для решения задачи использовать столько процессов (потоков), сколько слов в строке.

6. Задана строка S , имеющая следующий вид «число о число о число ... о число», где о может быть равно $+$, $-$, $/$, $*$. Вычислить выражение, записанное в строке. Входные данные: строка S .

7. Задана строка S , содержащая не менее двух предложений. Найти слово с максимальной длиной, встречающееся во всех предложениях, или сообщить, что такого слова нет. Входные данные: строка S .

8. Задана строка S , содержащая не менее двух слов, и символ k . Найти слово с минимальной длиной, начинающееся с символа k , или сообщить, что такого слова нет. Входные данные: строка S , символ k . Для решения задачи использовать столько процессов (потоков), сколько слов в строке.

9. Определить, является ли строка симметричной относительно указанного индекса r . Входные данные: строка S произвольной длины, целое число $r > 0$ и $<$ длины строки. Для решения задачи использовать два процесса (потока).

10. Задана строка S , и множество пар символов (a_i, b_i) $i = 1, 2 \dots n$, получить новую строку, заменив в строке S каждое вхождение a_i символа на b_i . Входные данные: строка S произвольной длины, целое положительное число n , множество пар символов (a_i, b_i) $i = 1, 2 \dots n$. Для решения задачи использовать четыре процесса (потока), разделив между ними строку S .

11. Задана строка S и множество пар символов (a_i, b_i) $i = 1, 2 \dots n$, получить новую строку, заменив в строке S каждое вхождение a_i символа на b_i . Входные данные: строка S произвольной длины, целое положительное число n , множество пар символов (a_i, b_i) $i = 1, 2 \dots n$. Для решения задачи использовать n процессов (потоков), работающих

параллельно, причем каждый процесс (поток) находит и заменяет только свою пару символов.

12. Задана строка S_1 , содержащая не менее двух слов, и строка S_2 , содержащая такое же количество слов, что и S_1 . Слово из строки S_2 является синонимом соответствующего слова из строки S_1 , если оно записано с префиксом '!'. Заменить в строке S_1 слова на их синонимы, удалив префиксы. Входные данные: строки S_1 и S_2 , содержащие одинаковое количество слов.

13. Найти среднее арифметическое всех факториалов в интервале $[A, B]$. Входные данные: положительное число A , положительное число B , целое число $k \geq 2$ и ≤ 6 . Использовать k процессов (потоков) для решения задачи. Предусмотреть возможность автоматического уменьшения числа процессов (потоков), если это целесообразно.

14. Сформировать одномерный массив B , элементами которого являются наибольшие из четырех рядом стоящих элементов, образующих квадрат 2×2 , в двоичном массиве A . Порядок занесения элементов в массив B не важен. Входные данные: целое положительное четное число n , массив чисел от A размерности $n \times n$.

15. Задана строка S , содержащая не менее двух слов, и символ k . Составить новую строку из тех слов строки S , которые начинаются и заканчиваются символом k . Учитывать порядок расположения слов в строке. Входные данные: строка S .

16. Сформировать одномерный массив B , элементами которого являются средние арифметические положительных элементов столбцов двумерного массива A . Найти сумму минимального и максимального элемента массива B . Порядок занесения элементов в массив B должен соответствовать их индексации в массиве A . Входные данные: целое положительное число n , целое положительное число k , массив чисел от A размерности $n \times k$. Использовать k или $k+1$ процессов (потоков) для решения задачи.

17. Сформировать массив B , заменяя элементы массива A их наибольшими делителями. Найти среднее арифметическое элементов массива B , сумма индексов которых является нечетным числом. Входные данные: целое положительное число n , массив чисел от A размерности $n \times n$, целое число $k \geq 2$ и $\leq n/2$. Использовать k процессов (потоков) для решения задачи.

18. Определить в каком числе - A или B больше вхождений цифры «0», вхождений цифры «1», вхождений цифры «2» ... вхождений

цифры «9». Определить число, сумма вхождений четных цифр которого больше. Входные данные: целое положительное число A , целое положительное число B . Для решения задачи использовать 10 процессов (потоков), каждый из которых должен определять количество вхождений своей цифры.

19. Задана строка S , содержащая не менее двух целых чисел, разделителем является запятая. Составить новую строку, записав в нее все нечетные числа в обратном порядке, пример: было 2961, стало 1692. Порядок занесения чисел в новую строку не важен. Входные данные: строка S , целое число $k \geq 2$ и ≤ 8 . Использовать k процессов (потоков) для решения задачи. Предусмотреть возможность автоматического уменьшения числа процессов (потоков), если это целесообразно.

20. Задан массив чисел, записанных в двоичной системе счисления, вычислить сумму всех чисел, не переводя их в десятичную систему счисления. Входные данные: целое положительное четное число $n > 2$, двоичный массив A размерности $n \times n$, заполненный нулями и единицами, элементы строк составляют числа, целое число $k \geq 2$ и $\leq n/2$. Использовать k процессов (потоков) для решения задачи.

Контрольные вопросы

1. Семафоры в UNIX. Отличие операций над UNIX-семафорами от классических операций.
2. Создание массива семафоров или доступ к уже существующему. Системный вызов `semget()`.
3. Семафоры для синхронизации процессов. Использование семафора для защиты критической секции.
4. Удаление набора семафоров из системы с помощью команды `ipcrm` или системного вызова `semctl()`.
5. Понятие о POSIX-семафорах.

Лабораторная работа 7

ОЧЕРЕДИ СООБЩЕНИЙ

Цели и задачи

Изучить механизм коммуникации процессов - сообщения. Научиться использовать очереди сообщений для организации взаимодействия процессов и их синхронизации.

7.1 Очереди сообщений в UNIX как составная часть System V IPC

Очередь сообщений представляет собой односторонний связанный список, расположенный в адресном пространстве ядра. Процессы могут записывать сообщения в очередь и изымать их из очереди. Само сообщение включает в себя тип сообщения – целое положительное число и непосредственно данные.

Так как очереди сообщений входят в состав средств System V IPC, для них верно все, что говорилось ранее об этих средствах в целом. Очереди сообщений, как и семафоры, и разделяемая память, являются средством связи с непрямой адресацией, требуют инициализации для организации взаимодействия процессов и специальных действий для освобождения системных ресурсов по окончании взаимодействия. Пространством имен очередей сообщений является то же самое множество значений ключа, генерируемых с помощью функции `ftok()`. Для выполнения примитивов `send` и `receive` соответствующим системным вызовам в качестве параметра передаются IPC-дескрипторы очередей сообщений, однозначно идентифицирующих их во всей вычислительной системе.

Очереди сообщений располагаются в адресном пространстве ядра операционной системы в виде односторонних списков и имеют ограничение по объему информации, хранящейся в каждой очереди. Каждый элемент списка представляет собой отдельное сообщение. Сообщения имеют атрибут, называемый типом сообщения.

Выборка сообщений из очереди (выполнение примитива `receive`) может осуществляться тремя способами:

- в порядке FIFO, независимо от типа сообщения.
- в порядке FIFO для сообщений конкретного типа.
- первым выбирается сообщение с минимальным типом, не

превышающим некоторого заданного значения, пришедшее раньше других сообщений с тем же типом.

Реализация примитивов `send` и `receive` обеспечивает скрытое от пользователя взаимoisключение во время помещения сообщения в очередь или его получения из очереди. Также она обеспечивает блокировку процесса при попытке выполнить примитив `receive` над пустой очередью или очередью, в которой отсутствуют сообщения запрошенного типа, или при попытке выполнить примитив `send` для очереди, в которой нет свободного места.

Очереди сообщений, как и другие средства System V IPC, позволяют организовать взаимодействие процессов, не находящихся одновременно в вычислительной системе.

7.2 Создание очереди сообщений или доступ к уже существующей. Системный вызов `msgget()`

Для создания очереди сообщений, ассоциированной с определенным ключом, или для доступа по ключу к уже существующей очереди используется системный вызов `msgget()`, являющийся аналогом системных вызовов `shmget()` для разделяемой памяти и `semget()` для массива семафоров, который возвращает значение IPC-дескриптора для этой очереди. При этом существуют те же способы создания и доступа, что и для разделяемой памяти или семафоров.

Системный вызов `msgget()`

Прототип системного вызова

```
#include <types.h>
```

```
#include <ipc.h>
```

```
#include <msg.h>
```

```
int msgget(key_t key, int msgflg);
```

Описание системного вызова

Системный вызов **`msgget`** предназначен для выполнения операции доступа к очереди сообщений и, в случае ее успешного завершения, возвращает дескриптор **System V IPC** для этой очереди (целое неотрицательное число, однозначно характеризующее очередь сообщений внутри вычислительной системы и использующееся в дальнейшем для других операций с ней).

Параметр **key** является ключом **System V IPC** для очереди сообщений, т. е. фактически ее именем из пространства имен **System V IPC**. В качестве значения этого параметра может быть использовано значение ключа, полученное с помощью функции **ftok()**, или специальное значение **IPC_PRIVATE**. Использование значения **IPC_PRIVATE** всегда приводит к попытке создания новой очереди сообщений с ключом, который не совпадает со значением ключа ни одной из уже существующих очередей и не может быть получен с помощью функции **ftok()** ни при одной комбинации ее параметров.

Параметр **msgflg** – флаги – играет роль только при создании новой очереди сообщений и определяет права различных пользователей при доступе к очереди, а также необходимость создания новой очереди и поведение системного вызова при попытке создания. Он является некоторой комбинацией (с помощью операции побитовое или – " | ") следующих предопределенных значений и восьмеричных прав доступа:

IPC_CREAT – если очереди для указанного ключа не существует, она должна быть создана;

IPC_EXCL – применяется совместно с флагом **IPC_CREAT**. При совместном их использовании и существовании массива с указанным ключом доступ к очереди не производится и констатируется ошибочная ситуация, при этом переменная **errno**, описанная в файле **<errno.h>**, примет значение **EEXIST**;

0400 – разрешено чтение для пользователя, создавшего очередь;

0200 – разрешена запись для пользователя, создавшего очередь;

0040 – разрешено чтение для группы пользователя, создавшего очередь;

0020 – разрешена запись для группы пользователя, создавшего очередь;

0004 – разрешено чтение для всех остальных пользователей;

0002 – разрешена запись для всех остальных пользователей;

Очередь сообщений имеет ограничение по общему количеству хранимой информации, которое может быть изменено администратором системы. Текущее значение ограничения можно узнать с помощью команды **ipcs -l**

Возвращаемое значение

Системный вызов возвращает значение дескриптора **System V IPC** для очереди сообщений при нормальном завершении и значение **-1** при возникновении ошибки.

7.3 Реализация примитивов send и receive.

Системные вызовы msgsnd() и msgrcv()

Для выполнения примитива **send** используется системный вызов **msgsnd()**, копирующий пользовательское сообщение в очередь сообщений, заданную IPC-дескриптором. При изучении описания этого вызова нужно обратить особое внимание на следующие моменты:

Тип данных `struct msgbuf` не является типом данных для пользовательских сообщений, а представляет собой лишь шаблон для создания таких типов. Пользователь сам должен создать структуру для своих сообщений, в которой первым полем должна быть переменная типа `long`, содержащая положительное значение типа сообщения.

В качестве третьего параметра – длины сообщения – указывается не вся длина структуры данных, соответствующей сообщению, а только длина полезной информации, т. е. информации, располагающейся в структуре данных после типа сообщения. Это значение может быть и равным 0 в случае, когда вся полезная информация заключается в самом факте прихода сообщения (сообщение используется как сигнальное средство связи).

В материалах семинаров мы, как правило, будем использовать нулевое значение флага системного вызова, которое приводит к блокировке процесса при отсутствии свободного места в очереди сообщений.

Системный вызов msgsnd()

Прототип системного вызова

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

```
#include <sys/ipc.h>
```

```
#include <sys/msg.h>
```

```
int msgsnd(int msqid, struct msgbuf *ptr,  
int length, int flag);
```

Описание системного вызова

Системный вызов **msgsnd** предназначен для помещения сообщения в очередь сообщений, т. е. является реализацией примитива send.

Параметр **msqid** является дескриптором **System V IPC** для очереди, в которую отправляется сообщение, т. е. значением, которое вернул системный вызов **msgget()** при создании очереди или при ее поиске по ключу.

Структура **struct msgbuf** описана в файле **<sys/msg.h>** как

```
struct msgbuf {  
    long mtype;  
    char mtext[1];};
```

Она представляет собой некоторый шаблон структуры сообщения пользователя. Сообщение пользователя – это структура, первый элемент которой обязательно имеет тип **long** и содержит тип сообщения, а далее следует информативная часть теоретически произвольной длины (практически в Linux она ограничена размером 4080 байт и может быть еще уменьшена системным администратором), содержащая собственно суть сообщения.

Например:

```
struct mymsgbuf {  
    long mtype;  
    char mtext[1024];  
    } mybuf;
```

При этом информация вовсе не обязана быть текстовой, например:

```
struct mymsgbuf {  
    long mtype;  
    struct {  
        int iinfo;  
        float finfo;  
    } info;  
    } mybuf;
```


Тип сообщения должен быть строго положительным числом. Действительная длина полезной части информации (т. е. информации, расположенной в структуре после типа сообщения) должна быть передана системному вызову в качестве параметра `length`. Этот параметр может быть равен и 0, если вся полезная информация заключается в самом факте наличия сообщения. Системный вызов копирует сообщение, расположенное по адресу, на который указывает параметр `ptr`, в очередь сообщений, заданную дескриптором `msqid`. Параметр `flag` может принимать два значения: 0 и `IPC_NOWAIT`. Если значение флага равно 0, и в очереди не хватает места для того, чтобы поместить сообщение, то системный вызов блокируется до тех пор, пока не освободится место. При значении флага `IPC_NOWAIT` системный вызов в этой ситуации не блокируется, а констатирует возникновение ошибки с установлением значения переменной `errno`, описанной в файле `<errno.h>`, равным `EAGAIN`.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Примитив **receive** реализуется системным вызовом **msgrcv()**. При изучении описания этого вызова нужно обратить особое внимание на следующие моменты:

Тип данных **struct msgbuf**, как и для вызова **msgsnd()**, является лишь шаблоном для пользовательского типа данных.

Способ выбора сообщения задается нулевым, положительным или отрицательным значением параметра **type**. Точное значение типа выбранного сообщения можно определить из соответствующего поля структуры, в которую системный вызов скопирует сообщение.

Системный вызов возвращает длину только полезной части скопированной информации, т. е. информации, расположенной в структуре после поля типа сообщения.

Выбранное сообщение удаляется из очереди сообщений.

В качестве параметра **length** указывается максимальная длина полезной части информации, которая может быть размещена в структуре, адресованной параметром `ptr`.

В материалах семинаров мы будем, как правило, пользоваться нулевым значением флагов для системного вызова, которое приводит

к блокировке процесса в случае отсутствия в очереди сообщений с запрошенным типом и к ошибочной ситуации в случае, когда длина информативной части выбранного сообщения превышает длину, специфицированную в параметре **length**.

Системный вызов **msgrcv()**

Прототип системного вызова

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

```
#include <sys/ipc.h>
```

```
#include <sys/msg.h>
```

```
int msgrcv(int msqid, struct msgbuf *ptr,  
int length, long type, int flag);
```

Описание системного вызова

Системный вызов **msgrcv** предназначен для получения сообщения из очереди сообщений, т. е. является реализацией примитива **receive**.

Параметр **msqid** является дескриптором **System V IPC** для очереди, из которой должно быть получено сообщение, т. е. значением, которое вернул системный вызов **msgget()** при создании очереди или при ее поиске по ключу.

Параметр **type** определяет способ выборки сообщения из очереди следующим образом

Таблица 7.1 Способ выборки сообщения из очереди

Способ выборки	Значение параметра type
В порядке FIFO, независимо от типа сообщения	0
В порядке FIFO для сообщений с типом n	n
Первым выбирается сообщение с минимальным типом, не превышающим значения n , пришедшее ранее всех других сообщений с тем же типом	- n

Максимально возможная длина информативной части сообщения в операционной системе Linux составляет 4080 байт и может быть уменьшена при генерации системы. Текущее значение максимальной длины можно определить с помощью команды

```
ipcs -l
```

7.4 Удаление очереди сообщений из системы с помощью команды `ipcrm` или системного вызова `msgctl()`

После завершения процессов, использовавших очередь сообщений, она не удаляется из системы автоматически, а продолжает сохраняться в системе вместе со всеми невостребованными сообщениями до тех пор, пока не будет выполнена специальная команда или специальный системный вызов. Для удаления очереди сообщений можно воспользоваться командой `ipcrm`, которая в этом случае примет вид:

`ipcrm msg <IPC идентификатор>`

Если какой-либо процесс находился в состоянии «ожидание» при выполнении системного вызова `msgrcv()` или `msgsnd()` для удаляемой очереди, то он будет разблокирован, и системный вызов констатирует наличие ошибочной ситуации.

Можно удалить очередь сообщений и с помощью системного вызова `msgctl()`.

Системный вызов `msgctl()`

Прототип системного вызова

`#include <sys/types.h>`

`#include <sys/ipc.h>`

`#include <sys/msg.h>`

**`int msgctl(int msqid, int cmd,
 struct msqid_ds *buf);`**

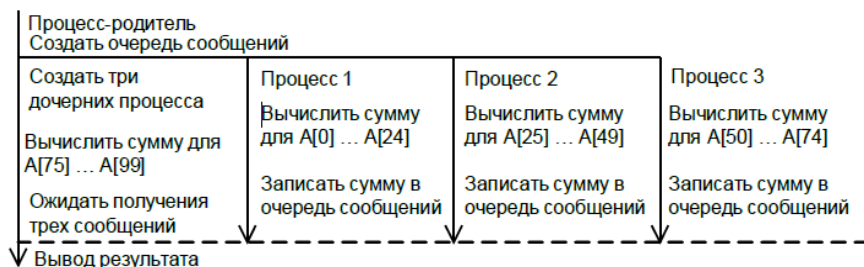
Описание системного вызова

Системный вызов **`msgctl`** предназначен для получения информации об очереди сообщений, изменения ее атрибутов и удаления из системы. Данное описание не является полным описанием системного вызова. Для изучения полного описания обращайтесь к **UNIX Manual**.

В нашем курсе мы будем пользоваться системным вызовом **msgctl** только для удаления очереди сообщений из системы. Параметр **msgid** является дескриптором **System V IPC** для очереди сообщений, т. е. значением, которое вернул системный вызов **msgget()** при создании очереди или при ее поиске по ключу.

Возвращаемое значение

7.5 Пример использования очереди сообщений



Программа 7-01, пример использования очереди сообщений

```
#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <iostream>
using namespace std;
int msgid; //для хранения дескриптора очереди сообщений
int A[100]; //массив, сумма элементов которого вычисляется
процессами
struct tmsg //структура для сообщений
{
    int mtype; //тип сообщения
    int mdata; //данные сообщения
}
m;
int summa (int p) //для вычисления суммы части элементов массива
{
    int i, sum = 0 ; //для суммирования элементов
    int begin = 25*p; //индекс массива, с которого начинается
суммирование
    int end = begin+25; //индекс массива, на котором завершается
суммирование
    for(i=begin; i<end; i++) sum+=A[i]; //вычисление суммы части
элементов массива
    m.mtype = 1; //установить тип сообщения в 1
    m.mdata=sum; //записать вычисленную сумму в сообщение
    msgsnd(msgid, &m, 2, 0); //послать сообщение в очередь, объем 2
байта
    return sum ; //возвратить вычисленную сумму
}
int main()
{
    //тут должна быть инициализация элементов массива A
    // создать очередь сообщений
    msgid = msgget(IPC_PRIVATE, IPC_CREAT|0666);
    if (msgid < 0 ) //если не удалось создать очередь сообщений,
завершить выполнение
```

```

{
cout<<"Ошибка"<<endl;
return 0;
}
for (int i=0 ; i<3 ; i++) //создать три процесса-потомка
{
if (fork() == 0 ) //истинно для дочернего процесса
{
summa(i);
return 1;
}
} //родительский процесс вычисляет последнюю четверть массива
int rez = summa(3);
for (int i=0 ; i<3 ; i++) //дождаться получения трех сообщений
{
msgrcv(msgid, &m, 2, 0, 0); //тип получаемого сообщения не важен
rez += m.mdata; //добавить данные сообщения к результату
} cout<<"Сумма = "<< rez<<endl; //вывести на экран сумму всех
элементов массива
return 1;
}

```

Листинг 7-01. Пример использования очереди сообщений

Программа 7-02а, пример с однонаправленной передачей текстовой информации.

Эта программа получает доступ к очереди сообщений, отправляет в нее 5 текстовых сообщений с типом 1 и одно пустое сообщение с типом 255, которое будет служить для программы 7-02b сигналом прекращения работы.

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <stdio.h>
#include <iostream>
using namespace std;
#define LAST_MESSAGE 255 /* Тип сообщения для прекращения

```

```

работы программы 2b */
int main()
{
    int msqid; /* IPC дескриптор для очереди сообщений */
    char pathname[] = "Имя файла"; /* Имя файла, используемое для
    генерации ключа. Файл с таким именем должен существовать в
    текущей директории */
    key_t key; /* IPC ключ */
    int i, len; /* Счетчик цикла и длина информативной части
    сообщения*/
    /* Ниже следует пользовательская структура для сообщения */
    struct mymsgbuf
    {
        long mtype;
        char mtext[81];
    }
    mybuf;
    /* Генерирование IPC ключа из имени файла 2a в текущей директории
    и номера экземпляра очереди сообщений 0. */
    if((key = ftok(pathname,0)) < 0)
    {
        cout<<"Can't generate key"<<endl;
        exit(-1);
    }
    /* Попытка получить доступ по ключу к очереди сообщений, если она
    существует, или создать ее, с правами доступа read & write для всех
    пользователей */
    if((msqid = msgget(key, 0666 | IPC_CREAT)) < 0)
    {
        cout<<"Can't get msqid "<< endl;
        exit(-1);
    }
    /* Посылка в цикле 5 сообщений с типом 1 в очередь сообщений,
    идентифицируемую msqid.*/
    for (i = 1; i <= 5; i++)
    {
        /* Сначала заполнение структуры для сообщения и определение длины
        информативной части */

```

```

mybuf.mtype = 1;
strcpy(mybuf.mtext, "This is text message");
len = strlen(mybuf.mtext)+1;
/* Отправка сообщения. В случае ошибки сообщение об этом и
удаление очереди сообщений из системы. */
if (msgsnd(msqid, (struct msgbuf *) &mybuf, len, 0) < 0)
{
    cout<<"Can't send message to queue"<< endl;
    msgctl(msqid, IPC_RMID, (struct msqid_ds *) NULL);
    exit(-1);
}
}
/* Отправка сообщения, которое заставит получающий процесс
прекратить работу, с типом LAST_MESSAGE и длиной 0 */
mybuf.mtype = LAST_MESSAGE;
len = 0;
if (msgsnd(msqid, (struct msgbuf *) &mybuf, len, 0) < 0)
{
    cout<<"Can't send message to queue "<< endl;
    msgctl(msqid, IPC_RMID, (struct msqid_ds *) NULL);
    exit(-1);
}
return 0;
}

```

Листинг 7-02а. Программа 6-02а для иллюстрации работы с очередями сообщений.

Программа 7-02б. Пример использования очереди сообщений.

Эта программа получает доступ к очереди сообщений и читает из нее сообщения с любым типом в порядке FIFO до тех пор, пока не получит сообщение с типом 255, которое будет служить сигналом прекращения работы.

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>

```



```

#include <stdio.h>
#include <iostream>
using namespace std;
#define LAST_MESSAGE 255 /* Тип сообщения для прекращения
работы */
int main()
{
    int msqid; /* IPC дескриптор для очереди сообщений */
    char pathname[] = "Имя файла"; /* Имя файла, используемое для
генерации ключа. Файл с таким именем должен существовать в
текущей директории */
    key_t key; /* IPC ключ */
    int len, maxlen; /* Реальная длина и максимальная длина
информативной части сообщения */
    /* Ниже следует пользовательская структура для сообщения */
    struct mymsgbuf
    {
        long mtype;
        char mtext[81];
    }
    mybuf;
    /* Генерирование IPC ключа из имени файла 2а в текущей
директории и номера экземпляра очереди сообщений 0 */
    if((key = ftok(pathname,0)) < 0)
    {
        cout<<"Can't generate key"<<endl;
        exit(-1);
    }
    /* Попытка получить доступ по ключу к очереди сообщений, если
она существует, или создать ее, с правами доступа read & write для
всех пользователей */
    if((msqid = msgget(key, 0666 | IPC_CREAT)) < 0)
    {
        cout<<"Can't get msqid"<<endl;
        exit(-1);
    }
    while(1)
    {

```

```

/* В бесконечном цикле принятие сообщения любого типа в порядке
FIFO с максимальной длиной информативной части 81 символ до тех
пор, пока не поступит сообщение с типом LAST_MESSAGE*/
    maxlen = 81;
    if(( len = msgrcv(msqid, (struct msghdr *) &mybuf, maxlen, 0, 0) < 0)
    {
cout<<"Can't receive message from queue"<<endl;
exit(-1);
    }
/* Если принятое сообщение имеет тип LAST_MESSAGE,
прекращение работы и удаление очереди сообщений из системы. В
противном случае вывод принятого сообщения. */
    if (mybuf.mtype == LAST_MESSAGE)
    {
        msgctl(msqid, IPC_RMID, (struct msqid_ds *) NULL);
        exit(0);
    }
    cout<<"message type =" << mybuf.mtype<< "info =" <<
mybuf.mtext<< endl;
    }
    return 0; /* Исключительно для отсутствия warning'ов при
компиляции. */
}

```

Листинг 7-02б. Программа 7-02б для иллюстрации работы с очередями сообщений

Первая из этих программ посылает пять текстовых сообщений с типом 1 и одно сообщение нулевой длины с типом 255 второй программе. Вторая программа в цикле принимает сообщения любого типа в порядке FIFO и печатает их содержимое до тех пор, пока не получит сообщение с типом 255. Сообщение с типом 255 служит для нее сигналом к завершению работы и ликвидации очереди сообщений. Если перед запуском любой из программ очередь сообщений еще отсутствовала в системе, то программа создаст ее. Необходимо обратить внимание на использование сообщения с типом 255 в качестве сигнала прекращения работы второго процесса. Это сообщение имеет нулевую длину, так как его информативность исчерпывается самим фактом наличия сообщения.

Модификация предыдущего примера для передачи числовой информации. В описании системных вызовов `msgsnd()` и `msgrcv()` говорится о том, что передаваемая информации не обязательно должна представлять собой текст. Можно воспользоваться очередями сообщений для передачи данных любого вида. При передаче разнородной информации целесообразно информативную часть объединять внутри сообщения в отдельную структуру, для правильного вычисления длины информативной части.

```
struct mymsgbuf {  
    long mtype;  
    struct {  
        short sinfo;  
        float finfo;  
    } info;  
} mybuf;
```

В некоторых вычислительных системах числовые данные размещаются в памяти с выравниванием на определенные адреса (например, на адреса, кратные 4).

Поэтому реальный размер памяти, необходимой для размещения нескольких числовых данных, может оказаться больше суммы длин этих данных, т. е. в нашем случае

`sizeof(info)>=sizeof(short)+sizeof(float)`

Для полной передачи информативной части сообщения в качестве длины нужно указывать не сумму длин полей, а полную длину структуры.

7.6 Понятие мультиплексирования. Мультиплексирование сообщений. Модель взаимодействия процессов клиент-сервер. Неравноправность клиента и сервера

Используя технику из предыдущего примера, можно организовать получение сообщений одним процессом от множества других процессов через одну очередь сообщений и отправку им ответов через ту же очередь сообщений, т. е. осуществить мультиплексирование сообщений. Вообще под мультиплексированием

информации понимают возможность одновременного обмена информацией с несколькими партнерами. Метод мультиплексирования широко применяется в модели взаимодействия процессов клиент-сервер. В этой модели один из процессов является сервером. Сервер получает запросы от других процессов – клиентов – на выполнение некоторых действий и отправляет им результаты обработки запросов. Чаще всего модель клиент-сервер используется при разработке сетевых приложений. Она изначально предполагает, что взаимодействующие процессы неравноправны: Сервер, как правило, работает постоянно, на всем протяжении жизни приложения, а клиенты могут работать эпизодически.

Сервер ждет запроса от клиентов, инициатором же взаимодействия является клиент.

Как правило, клиент обращается к одному серверу за раз, в то время как к серверу могут одновременно поступать запросы от нескольких клиентов.

Клиент должен знать, как обратиться к серверу (например, какого типа сообщения он воспринимает) перед началом организации запроса к серверу, в то время как сервер может получить недостающую информацию о клиенте из прошедшего запроса.

Пример схемы мультиплексирования сообщений через одну очередь сообщений для модели клиент-сервер.

Пусть сервер получает из очереди сообщений только сообщения с типом 1. В состав сообщений с типом 1, посылаемых серверу, процессы-клиенты включают значения своих идентификаторов процесса. Приняв сообщение с типом 1, сервер анализирует его содержание, выявляет идентификатор процесса, пославшего запрос, и отвечает клиенту, посылая сообщение с типом, равным идентификатору запрашивавшего процесса. Процесс-клиент после отправления запроса ожидает ответа в виде сообщения с типом, равным своему идентификатору. Поскольку идентификаторы процессов в системе различны, и ни один пользовательский процесс не может иметь PID, равный 1, все сообщения могут быть прочитаны только теми процессами, которым они адресованы. Если обработка запроса занимает продолжительное время, сервер может организовывать параллельную обработку запросов, порождая для каждого запроса новый процесс-ребенок или новую нить исполнения.

Порядок выполнения лабораторной работы

1. Разработать алгоритм решения задания с учетом разделения вычислений между несколькими процессами. Для обмена информацией между процессами использовать очередь сообщений.

2. Реализовать алгоритм решения задания и протестировать его на нескольких примерах.

3. Посмотреть в динамике работу семафоров для созданного приложения, используя команду `ipcs -q`.

4. Наберите программы, сохраните под именами 7-1a.c и 7-1b.c соответственно, откомпилируйте и проверьте правильность их поведения.

5. Модифицируйте программы 7-1a.c и 7-1b.c для передачи нетекстовых сообщений.

6. Напишите, откомпилируйте и прогоните программы, осуществляющие двустороннюю связь через одну очередь сообщений.

7. Напишите, откомпилируйте и прогоните программы сервера и клиентов для предложенной схемы мультимплексирования сообщений.

8. Задача повышенной сложности: реализуйте семафоры через очереди сообщений.

Варианты индивидуальных заданий

1. Сформировать массив, элементами которого являются целые числа больше A , меньше B и не равные C_1, C_2, \dots, C_n . Найти среднее арифметическое элементов полученного массива. Входные данные: число A , число B , произвольное количество чисел C_1, C_2, \dots, C_n . Использовать не менее четырех процессов для решения задачи.

2. Из трех матриц A, B, C определить матрицу с наибольшим определителем. Найти сумму элементов для выбранной матрицы. Входные данные: массивы чисел A, B, C размерности 4×4 . Использовать не менее трех процессов для решения задачи.

3. Из элементов массива найти все пары чисел (a, b) , где a равно сумме всех делителей числа b , за исключением единицы и самого b . Найти пару с максимальным значением $a+b$. Входные данные: целое положительное число n , целое положительное число $k > 1$, массив целых положительных чисел A размерности n .

Использовать k процессов для решения задачи. Предусмотреть возможность автоматического уменьшения числа процессов, если это целесообразно.

4. Из элементов массива найти все пары чисел (a, b) , где $b-a=k$. Найти пару с максимальным значением $|a|*|b|$. Входные данные: целое положительное число $n>3$, целое число k , массив целых чисел A размерности n . Использовать не менее четырех процессов для решения задачи.

5. Найти сумму всех элементов массива A , которые являются числами Армстронга. Натуральное число называется числом Армстронга, когда сумма его цифр, возведенная в степень равную числу этих цифр, равна самому числу. Входные данные: целое положительное число n , целое положительное число $k>1$, массив натуральных чисел A размерности n . Использовать k процессов для решения задачи. Предусмотреть возможность автоматического уменьшения числа процессов, если это целесообразно.

6. Найти все четырехзначные числа $abcd$, для которых $a+b+c+d = k$ или $a-b+c-d = k$ или $ab - cd = k$. Вычислить сумму найденных чисел. Входные данные: целое положительное число k . Использовать девять процессов для решения задачи, где каждый процесс работает со своим числовым интервалом.

7. Найти все четырехзначные числа $abcd$, для которых $a+b+c+d = k$ или $a*b*c*d = k$ или $ac - bd = k$. Вычислить среднее арифметическое найденных чисел. Входные данные: целое положительное число k . Использовать три процесса для решения задачи, где каждый процесс вычисляет собственное условие, из трех заданных.

8. Найти максимальный элемент в матрице A и поменять его местами с максимальным элементом матрицы B . Затем вычислить сумму определителей полученных матриц. Входные данные: массивы чисел A, B размерности 4×4 .

9. Найти все n -значные числа, цифры которых образуют убывающую (960) или возрастающую (1258) последовательность. Вычислить среднее арифметическое найденных чисел. Входные данные: целое положительное число $n<11$. Использовать n или $n+1$ процессов для решения задачи.

10. Найти все n -значные числа, делящиеся нацело на каждую из своих цифр. Вычислить среднее арифметическое найденных чисел.

Входные данные: целое положительное число $n < 11$. Использовать девять процессов для решения задачи, где каждый процесс работает со своим числовым интервалом.

11. Из элементов массива A найти все числа, равные $n \cdot (n+1) \cdot (n+2)$, где n – любое натуральное число. Вычислить сумму максимального и минимального найденного числа. Входные данные: целое положительное число k , массив натуральных чисел A размерности k .

12. Найти процент счастливых билетов с шестизначными номерами из интервала (A, B) . Входные данные: шестизначное число A , шестизначное число $B > A$, целое положительное число $k > 1$. Использовать k процессов для решения задачи. Предусмотреть возможность автоматического уменьшения числа процессов, если это целесообразно.

13. Найти все n -значные числа, содержащие k единиц и не содержащие нулей. Вычислить среднее арифметическое найденных чисел. Входные данные: целое положительное число $n < 11$, целое неотрицательное число $k \leq n$. Предусмотреть обработку ситуаций, когда порождение дочерних процессов является излишним.

14. Сформировать новый массив из соответствующих элементов массива A , отняв от каждого элемента сумму его цифр, от получившегося числа снова отнять сумму его цифр и так далее k раз. Найти сумму минимального элемента массива A и максимального элемента сформированного массива. Входные данные: целое положительное число k , целое положительное число n , массив целых чисел A размерности n .

15. Вычислить сумму тех целых чисел из интервала (A, B) , которые равны двойке в произвольной целой степени. Входные данные: натуральное число A , натуральное число $B > A$.

16. Найти все числа из элементов массива A , равные произведению двух произвольных простых чисел. Вычислить сумму из k наибольших найденных чисел. Входные данные: целое положительное число k , целое положительное число n , массив целых чисел A размерности n .

17. Из множества векторов найти вектор с максимальной длиной и вектор с минимальной длиной, вычислить векторное и скалярное произведения найденных векторов. Входные данные: целое положительное число n , массив целых чисел A размерности $n \times 3$, где

каждый вектор записан в строке. Использовать два процесса для решения задачи.

18. Вычислить сумму тех целых чисел из интервала (A, B), которые равны $n + \text{произвольное простое число}$. Входные данные: натуральное число A, натуральное число $B > A$, целое число n, натуральное число $k > 1$. Использовать k процессов для решения задачи. Предусмотреть возможность автоматического уменьшения числа процессов, если это целесообразно.

19. Из трех матриц A, B, C определить матрицу, содержащую максимальный элемент, и матрицу, содержащую минимальный элемент. Вычислить произведение двух найденных матриц. Входные данные: массивы чисел A, B, C размерности 3×3 . Использовать не менее трех процессов для решения задачи.

20. Найти количество вхождений в строку S, каждого символа, в ней содержащегося. Определить наиболее часто повторяющийся символ. Входные данные: строка S.

Контрольные вопросы

1. Описать механизм коммуникации процессов – сообщения.
2. Очереди сообщений в UNIX как составная часть System V IPC.
3. Создание очереди сообщений или доступ к уже существующей. Системный вызов `msgget()`.
4. Реализация примитивов `send` и `receiv`. Системные вызовы `msgsnd()` и `msgrcv()`.
5. Удаление очереди сообщений из системы с помощью команды `ipcrm` или системного вызова `msgctl()`.
6. Понятие мультиплексирования. Мультиплексирование сообщений.
7. Модель взаимодействия процессов клиент-сервер. Неравноправность клиента и сервера.

Лабораторная работа 8

ОРГАНИЗАЦИЯ ФАЙЛОВОЙ СИСТЕМЫ В UNIX.

РАБОТА С ФАЙЛАМИ И ДИРЕКТОРИЯМИ. ПОНЯТИЕ О MEMORY MAPPED.

Цели и задачи

Изучение организации файловой системы UNIX. Знакомство с командами и системными вызовами, используемыми для работы с файлами и директориями. Рассматривается понятие memory mapped файлы.

8.1 Разделы носителя информации (partitions) в UNIX

Физические носители информации – магнитные или оптические диски, ленты и т.д., использующиеся как физическая основа для хранения файлов, в операционных системах принято логически делить на разделы (partitions) или логические диски. Причем слово «делить» не следует понимать буквально, в некоторых системах несколько физических дисков могут быть объединены в один раздел.

В операционной системе UNIX физический носитель информации обычно представляет собой один или несколько разделов. В большинстве случаев разбиение на разделы производится линейно, хотя некоторые варианты UNIX могут допускать некое подобие древовидного разбиения (Solaris). Количество разделов и их размеры определяются при форматировании диска.

Наличие нескольких разделов на диске может определяться требованиями операционной системы или пожеланиями пользователя. Если пользователь хочет разместить на одном жестком диске несколько операционных систем с возможностью попеременной работы в них, тогда он размещает каждую операционную систему в своем разделе. Или другая ситуация: необходимость работы с несколькими видами файловых систем. Под каждый тип файловой системы выделяется отдельный логический диск. Третий вариант – это разбиение диска на разделы для размещения в разных разделах различных категорий файлов.

Примером операционной системы, внутренние требования которой приводят к появлению нескольких разделов на диске, могут

служить ранние версии MS-DOS, для которых максимальный размер логического диска не превышал 32 Мбайт.

Для простоты далее в этих семинарах будем полагать, что у нас имеется только один раздел и, следовательно, одна файловая система.

8.2 Логическая структура файловой системы и типы файлов в UNIX

Файл — именованный абстрактный объект, обладающий определенными свойствами. При этом в пространстве имен файлов одному файлу могут соответствовать несколько имен.

В операционной системе UNIX существуют файлы нескольких типов, а именно:

- обычные или регулярные файлы;
- директории или каталоги;
- файлы типа FIFO или именованные рір'ы;
- специальные файлы устройств;
- сокеты (sockets);
- специальные файлы связи (link).

Файлы всех перечисленных типов логически объединены в ациклический граф с однонаправленными ребрами, получающийся из дерева в результате сращивания нескольких терминальных узлов дерева или нескольких его нетерминальных узлов таким образом, чтобы полученный граф не содержал циклов. В нетерминальных узлах такого ациклического графа (т. е. в узлах, из которых выходят ребра) могут располагаться только файлы типов «директория» и «связь». Причем из узла, в котором располагается файл типа «связь», может выходить только ровно одно ребро. В терминальных узлах этого ациклического графа (т. е. в узлах, из которых не выходит ребер) могут располагаться файлы любых типов (рис. 8.1), хотя присутствие в терминальном узле файла типа «связь» обычно говорит о некотором нарушении целостности файловой системы.

В отличие от древовидной структуры набора файлов, где имена файлов связывались с узлами дерева, в таком ациклическом графе имя файла связывается не с узлом, соответствующим файлу, а с входящим в него ребром. Ребра, выходящие из узлов, соответствующих файлам типа «связь», являются неименованными.

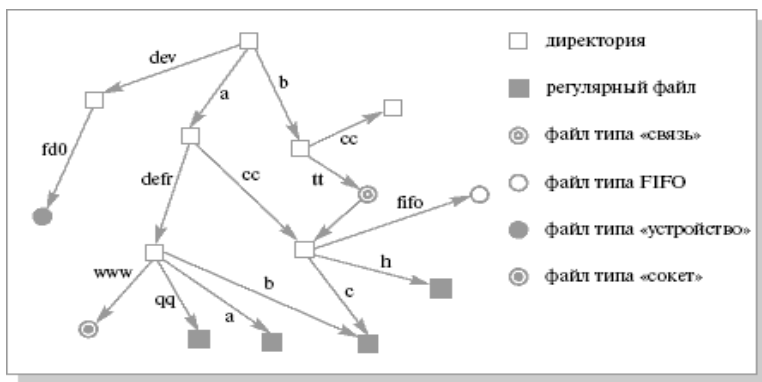


Рис. 8.1. Пример графа файловой системы

Надо отметить, что практически во всех существующих реализациях UNIX-подобных систем в узел графа, соответствующий файлу типа «директория», не может входить более одного именованного ребра, хотя стандарт на операционную систему UNIX и не запрещает этого. В качестве полного имени файла может использоваться любое имя, получающееся при прохождении по ребрам от корневого узла графа (т. е. узла, в который не входит ни одно ребро) до узла, соответствующего этому файлу, по любому пути с помощью следующего алгоритма:

1. Если интересующему нас файлу соответствует корневой узел, то файл имеет имя "/".
2. Берем первое именованное ребро в пути и записываем его имя, которому предворяем символ "/".
3. Для каждого очередного именованного ребра в пути приписываем к уже получившейся строке справа символ "/" и имя соответствующего ребра.

Полное имя является уникальным для всей файловой системы и однозначно определяет соответствующий ему файл.

8.3 Организация файла на диске в UNIX на примере файловой системы s5fs. Понятие индексного узла (inode)

Все дисковое пространство раздела в файловой системе s5fs (System V file system) логически разделяется на две части: заголовок раздела и логические блоки данных. Заголовок раздела содержит служебную информацию, необходимую для работы файловой

системы, и обычно располагается в самом начале раздела. Логические блоки хранят собственно содержательную информацию файлов и часть информации о размещении файлов на диске (т. е. какие логические блоки и в каком порядке содержат информацию, записанную в файл). Для размещения любого файла на диске используется метод индексных узлов (inode – от index node). Индексный узел содержит атрибуты файла и оставшуюся часть информации о его размещении на диске. Такие типы файлов, как «связь», «сокет», «устройство», «FIFO» не занимают на диске никакого иного места, кроме индексного узла (им не выделяется логических блоков). Все необходимое для работы с этими типами файлов содержится в их атрибутах. Часть атрибутов файлов, хранящихся в индексном узле и свойственных большинству типов файлов:

- Тип файла и права различных категорий пользователей для доступа к нему.
- Идентификаторы владельца-пользователя и владельца-группы.
- Размер файла в байтах (только для регулярных файлов, директорий и файлов типа «связь»).
- Время последнего доступа к файлу.
- Время последней модификации файла.
- Время последней модификации самого индексного узла.

Количество индексных узлов в разделе является постоянной величиной, определяемой на этапе генерации файловой системы. Все индексные узлы системы организованы в виде массива, хранящегося в заголовке раздела. Каждому файлу соответствует только один элемент этого массива и, наоборот, каждому непустому элементу этого массива соответствует только один файл. Таким образом, каждый файл на диске может быть однозначно идентифицирован номером своего индексного узла (его индексом в массиве). На языке представления логической организации файловой системы в виде графа это означает, что каждому узлу графа соответствует только один номер индексного узла, и никакие два узла графа не могут иметь одинаковые номера. Надо отметить, что свойством уникальности номеров индексных узлов, идентифицирующих файлы, мы уже неявно пользовались при работе с именованными `pip`'ами и средствами System V IPC. Для именованного `pip`'а именно номер индексного узла, соответствующего файлу с типом FIFO, является той самой точкой

привязки, пользуясь которой, неродственные процессы могут получить данные о расположении `rip'a` в адресном пространстве ядра и его состоянии и связаться друг с другом. Для средств System V IPC при генерации IPC-ключа с помощью функции `flok()` в действительности используется не имя заданного файла, а номер соответствующего ему индексного дескриптора, который по определенному алгоритму объединяется с номером экземпляра средства связи.

8.4 Организация директорий (каталогов) в UNIX

Содержимое регулярных файлов (информация, находящаяся в них, и способ ее организации) всецело определяется программистом, создающим файл. В отличие от регулярных, остальные типы файлов, содержащих данные, т. е. директории и связи, имеют жестко заданную структуру и содержание, определяемые типом используемой файловой системы. Основным содержимым файлов типа «директория», являются имена файлов, лежащих непосредственно в этих директориях, и соответствующие им номера индексных узлов. В терминах представления в виде графа содержимое директорий представляет собой имена ребер, выходящих из узлов, соответствующих директориям, вместе с индексными номерами узлов, к которым они ведут.

В файловой системе **s5fs** пространство имен файлов (ребер) содержит имена длиной не более 14 символов, а максимальное количество **inode** в одном разделе файловой системы не может превышать значения 65535. Эти ограничения не позволяют давать файлам осмысленные имена и приводят к необходимости разбиения больших жестких дисков на несколько разделов. Зато они помогают упростить структуру хранения информации в директории. Все содержимое директории представляет собой таблицу, в которой каждый элемент имеет фиксированный размер в 16 байт. Из них 14 байт отводится под имя соответствующего файла (ребра), а 2 байта – под номер его индексного узла. При этом первый элемент таблицы дополнительно содержит ссылку на саму данную директорию под именем " .", а второй элемент таблицы – ссылку на родительский каталог (если он существует), т. е. на узел графа, из которого выходит единственное именованное ребро, ведущее к текущему узлу, под именем " ..". В

более современной файловой системе **FFS (Fast File System)** размерность пространства имен файлов (ребер) увеличена до 255 символов. Это позволило использовать практически любые мыслимые имена для файлов (вряд ли найдется программист, которому будет не лень набирать для имени более 255 символов), но пришлось изменить структуру каталога (чтобы уменьшить его размеры и не хранить пустые байты). В системе **FFS** каталог представляет собой таблицу из записей переменной длины. В структуру каждой записи входят: номер индексного узла, длина этой записи, длина имени файла и собственно его имя. Две первых записи в каталоге, как и в *s5fs*, по-прежнему адресуют саму данную директорию и ее родительский каталог.

8.5 Понятие суперблока

В предыдущих разделах уже была рассмотрена часть заголовка раздела. Оставшуюся часть заголовка в *s5fs* принято называть суперблоком. Суперблок хранит информацию, необходимую для правильного функционирования файловой системы в целом. В нем содержатся, в частности, следующие данные:

- Тип файловой системы.
- Флаги состояния файловой системы.
- Размер логического блока в байтах (обычно кратен 512 байтам).
- Размер файловой системы в логических блоках (включая сам суперблок и массив *inode*).
- Размер массива индексных узлов (т. е. сколько файлов может быть размещено в файловой системе).
- Число свободных индексных узлов (сколько файлов еще можно создать).
- Число свободных блоков для размещения данных.
- Часть списка свободных индексных узлов.
- Часть списка свободных блоков для размещения данных.

В некоторых модификациях файловой системы *s5fs* последние два списка выносятся за пределы суперблока, но остаются в заголовке раздела. При первом же обращении к файловой системе суперблок обычно целиком считывается в адресное пространство ядра для ускорения последующих обращений. Поскольку количество логических блоков и индексных узлов в файловой системе может быть

весьма большим, нецелесообразно хранить списки свободных блоков и узлов в суперблоке полностью. При работе с индексными узлами часть списка свободных узлов, находящаяся в суперблоке, постепенно убывает. Когда список почти исчерпан, операционная система сканирует массив индексных узлов и заново заполняет список. Часть списка свободных логических блоков, лежащая в суперблоке, содержит ссылку на продолжение списка, расположенное где-либо в блоках данных. Когда эта часть оказывается использованной, операционная система загружает на освободившееся место продолжение списка, а блок, применявшийся для его хранения, переводится в разряд свободных.

8.6 Операции над файлами и директориями

Хотя с точки зрения пользователя рассмотрение операций над файлами и директориями представляется достаточно простым и сводится к перечислению ряда системных вызовов и команд операционной системы, попытка систематического подхода к набору операций вызывает определенные затруднения. Далее речь пойдет в основном о регулярных файлах и файлах типа «директория». Существует два основных вида файлов, различающихся по методу доступа: файлы последовательного доступа и файлы прямого доступа. Если рассматривать файлы прямого и последовательного доступа как абстрактные типы данных, то они представляются как нечто, содержащее информацию, над которой можно совершать следующие операции:

Для последовательного доступа: чтение очередной порции данных (read), запись очередной порции данных (write) и позиционирование на начале файла (rewind).

Для прямого доступа: чтение очередной порции данных (read), запись очередной порции данных (write) и позиционирование на требуемой части данных (seek). Работа с объектами этих абстрактных типов подразумевает наличие еще двух необходимых операций: создание нового объекта (new) и уничтожение существующего объекта (free).

Расширение математической модели файла за счет добавления к хранимой информации атрибутов, присущих файлу (права доступа, учетные данные), влечет за собой появление еще двух операций:

прочитать атрибуты (get attribute) и установить их значения (set attribute). Наделение файлов какой-либо внутренней структурой (как у файла типа «директория») или наложение на набор файлов внешней логической структуры (объединение в ациклический направленный граф) приводит к появлению других наборов операций, составляющих интерфейс работы с файлами, которые, тем не менее, будут являться комбинациями перечисленных выше базовых операций.

Для директории, например, такой набор операций, определяемый ее внутренним строением, может выглядеть так: операции new, free, set attribute и get attribute остаются без изменений, а операции read, write и rewind (seek) заменяются более высокоуровневыми:

прочитать запись, соответствующую имени файла, – get record;

добавить новую запись – add record;

удалить запись, соответствующую имени файла, – delete record.

Неполный набор операций над файлами, связанный с их логическим объединением в структуру директорий, будет выглядеть следующим образом:

Операции для работы с атрибутами файлов – get attribute, set attribute.

Операции для работы с содержимым файлов – read, write, rewind(seek) для регулярных файлов и get record, add record, delete record для директорий.

Операция создания регулярного файла в некоторой директории (создание нового узла графа и добавление в граф нового именованного ребра, ведущего в этот узел из некоторого узла, соответствующего директории) – create. Эту операцию можно рассматривать как суперпозицию двух операций: базовой операции new для регулярного файла и add record для соответствующей директории.

Операция создания поддиректории в некоторой директории – make directory. Эта операция отличается от предыдущей операции create занесением в файл новой директории информации о файлах с именами "." и "..", т.е. по сути дела она есть суперпозиция операции create и двух операций add record.

Операция создания файла типа "связь" – symbolic link.

Операция создания файла типа "FIFO" – make FIFO.

Операция добавления к графу нового именованного ребра, ведущего от узла, соответствующего директории, к узлу, соответствующему любому другому типу файла, – link. Это просто add record с некоторыми ограничениями.

Операция удаления файла, не являющегося директорией или «связью» (удаление именованного ребра из графа, ведущего к терминальной вершине с одновременным удалением этой вершины, если к ней не ведут другие именованные ребра), – unlink.

Операция удаления файла типа «связь» (удаление именованного ребра, ведущего к узлу, соответствующему файлу типа «связь», с одновременным удалением этого узла и выходящего из него неименованного ребра, если к этому узлу не ведут другие именованные ребра), – unlink link.

Операция рекурсивного удаления директории со всеми входящими в нее файлами и поддиректориями – remove directory.

Операция переименования файла (ребра графа) – rename.

Операция перемещения файла из одной директории в другую (перемещается точка выхода именованного ребра, которое ведет к узлу, соответствующему данному файлу) – move.

Возможны и другие подобные операции.

Способ реализации файловой системы в реальной операционной системе также может добавлять новые операции. Если часть информации файловой системы или отдельного файла кэшируется в адресном пространстве ядра, то появляются операции синхронизации данных в кэше и на диске для всей системы в целом (sync) и для отдельного файла (sync file).

Все перечисленные операции могут быть выполнены процессом только при наличии у него определенных полномочий (прав доступа и т. д.). Для выполнения операций над файлами и директориями операционная система предоставляет процессам интерфейс в виде системных вызовов, библиотечных функций и команд операционной системы.

8.7 Системные вызовы и команды для выполнения операций над файлами и директориями

Далее в этом разделе, если не будет оговорено особо, под словом «файл» будет подразумеваться регулярный файл. Вся информация об атрибутах файла и его расположении на физическом носителе содержится в соответствующем файлу индексном узле и, возможно, в нескольких связанных с ним логических блоках. Чтобы при каждой операции над файлом не считывать эту информацию с

физического носителя заново, представляется логичным, считав информацию один раз при первом обращении к файлу, хранить ее в адресном пространстве процесса или в части адресного пространства ядра, характеризующей данный процесс.

С точки зрения пользовательского процесса каждый файл представляет собой линейный набор байт, снабженный указателем текущей позиции процесса в этом наборе. Все операции чтения из файла и записи в файл производятся в этом наборе с того места, на которое показывает указатель текущей позиции. По завершении операции чтения или записи указатель текущей позиции помещается после конца прочитанного или записанного участка файла. Значение этого указателя является динамической характеристикой файла для использующего его процесса и также должно храниться в РСВ.

На самом деле организация информации, описывающей открытые файлы в адресном пространстве ядра операционной системы UNIX, является более сложной. Некоторые файлы могут использоваться одновременно несколькими процессами независимо друг от друга или совместно. Для того чтобы не хранить дублирующуюся информацию об атрибутах файлов и их расположении на внешнем носителе для каждого процесса отдельно, такие данные обычно размещаются в адресном пространстве ядра операционной системы в единственном экземпляре, а доступ к ним процессы получают только при выполнении соответствующих системных вызовов для операций над файлами.

Независимое использование одного и того же файла несколькими процессами в операционной системе UNIX предполагает возможность для каждого процесса совершать операции чтения и записи в файл по своему усмотрению. При этом для корректной работы с информацией необходимо организовывать взаимоисключения для операций ввода-вывода. Совместное использование одного и того же файла в операционной системе UNIX возможно для близко родственных процессов, т. е. процессов, один из которых является потомком другого или которые имеют общего родителя. При совместном использовании файла процессы разделяют некоторые данные, необходимые для работы с файлом, в частности, указатель текущей позиции. Операции чтения или записи, выполненные в одном процессе, изменяют значение указателя текущей позиции во всех близко родственных процессах,

одновременно использующих этот файл.

Вся информация о файле, необходимая процессу для работы с ним, может быть разбита на три части:

- данные, специфичные для этого процесса;
- данные, общие для близко родственных процессов, совместно использующих файл, например, указатель текущей позиции;
- данные, являющиеся общими для всех процессов, использующих файл, – атрибуты и расположение файла.

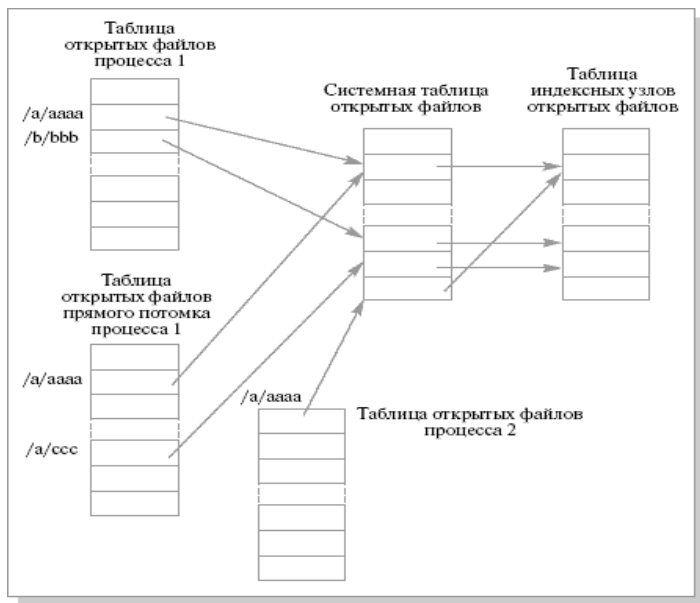


Рис. 8.2. Взаимосвязи между таблицами, содержащими данные об открытых файлах в системе

Для хранения этой информации применяются три различные связанные структуры данных, лежащие, как правило, в адресном пространстве ядра операционной системы, – таблица открытых файлов процесса, системная таблица открытых файлов и таблица индексных узлов открытых файлов. Для доступа к этой информации в управляющем блоке процесса заводится таблица открытых файлов, каждый непустой элемент которой содержит ссылку на

соответствующий элемент системной таблицы открытых файлов, содержащей данные, необходимые для совместного использования файла близко родственными процессами. Из системной таблицы открытых файлов можно по ссылке добраться до общих данных о файле, содержащихся в таблице индексных узлов открытых файлов (рис. 8.2). Только таблица открытых файлов процесса входит в состав его PCB и, соответственно, наследуется при рождении нового процесса. Индекс элемента в этой таблице (небольшое целое неотрицательное число) или файловый дескриптор является той величиной, характеризующей файл, которой может оперировать процесс при работе на уровне пользователя. В эту же таблицу открытых файлов помещаются и ссылки на данные, описывающие другие потоки ввода-вывода, такие как pipe и FIFO

Системный вызов `open()`. Для выполнения большинства операций над файлами через системные вызовы пользовательский процесс обычно должен указать в качестве одного из параметров системного вызова дескриптор файла, над которым нужно совершить операцию. Поэтому, прежде чем совершать операции, необходимо поместить информацию о файле в таблицы файлов и определить соответствующий файловый дескриптор. Для этого применяется процедура открытия файла, осуществляемая системным вызовом **`open()`**. При открытии файла операционная система проверяет, соответствуют ли права, которые запросил процесс для операций над файлом, правам доступа, установленным для этого файла. В случае соответствия она помещает необходимую информацию в системную таблицу файлов и, если этот файл не был ранее открыт другим процессом, в таблицу индексных дескрипторов открытых файлов. Далее операционная система находит пустой элемент в таблице открытых файлов процесса, устанавливает необходимую связь между всеми тремя таблицами и возвращает на пользовательский уровень дескриптор этого файла.

С помощью операции открытия файла операционная система осуществляет отображение из пространства имен файлов в дисковое пространство файловой системы, подготавливая почву для выполнения других операций.

Системный вызов `close()`. Обратным системным вызовом по отношению к системному вызову **`open()`** является системный вызов **`close()`**. После завершения работы с файлом процесс освобождает

выделенные ресурсы операционной системы и, возможно, синхронизирует информацию о файле, содержащуюся в таблице индексных узлов открытых файлов, с информацией на диске, используя этот системный вызов. Надо отметить, что место в таблице индексных узлов открытых файлов не освобождается по системному вызову `close()` до тех пор, пока в системе существует хотя бы один процесс, использующий этот файл. Для обеспечения такого поведения в ней для каждого индексного узла заводится счетчик числа открытий, увеличивающийся на 1 при каждом системном вызове `open()` для данного файла и уменьшающийся на 1 при каждом его закрытии. Очищение элемента таблицы индексных узлов открытых файлов с окончательной синхронизацией данных в памяти и на диске происходит только в том случае, если при очередном закрытии файла этот счетчик становится равным 0.

Операция создания файла. Системный вызов `creat()`

Прототип системного вызова

```
#include <fcntl.h>
```

```
int creat(char *path, int mode);
```

Описание системного вызова

Системный вызов `creat` эквивалентен системному вызову `open()` с параметром **flags**, установленным в значение **O_CREAT | O_WRONLY | O_TRUNC**.

Параметр **path** является указателем на строку, содержащую полное или относительное имя файла.

Если файла с указанным именем не существовало к моменту системного вызова, он будет создан и открыт только для выполнения операций записи. Если файл уже существовал, то он открывается также только для операции записи, при этом его длина уменьшается до 0 с одновременным сохранением всех других атрибутов файла.

Параметр **mode** устанавливает атрибуты прав доступа различных категорий пользователей к новому файлу при его создании. Этот параметр задается как сумма следующих восьмеричных значений:

0400 – разрешено чтение для пользователя, создавшего файл.

0200 – разрешена запись для пользователя, создавшего файл.

0100 – разрешено исполнение для пользователя, создавшего файл.

0040 – разрешено чтение для группы пользователя, создавшего файл.
0020 – разрешена запись для группы пользователя, создавшего файл.
0010 – разрешено исполнение для группы пользователя, создавшего файл.
0004 – разрешено чтение для всех остальных пользователей
0002 – разрешена запись для всех остальных пользователей
0001 – разрешено исполнение для всех остальных пользователей
При создании файла реально устанавливаемые права доступа получаются из стандартной комбинации параметра **mode** и маски создания файлов текущего процесса **umask**, а именно – они равны **mode & ~umask**.

Возвращаемое значение

Системный вызов возвращает значение файлового дескриптора для открытого файла при нормальном завершении и значение -1 при возникновении ошибки.

Системные вызовы для чтения атрибутов файла

Прототипы системных вызовов

```
#include <sys/stat.h>
#include <unistd.h>
int stat(char *filename,
         struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(char *filename,
         struct stat *buf);
```

Описание системных вызовов

Системные вызовы **stat**, **fstat** и **lstat** служат для получения информации об атрибутах файла.

Системный вызов **stat** читает информацию об атрибутах файла, на имя которого указывает параметр **filename**, и заполняет ими структуру, расположенную по адресу **buf**. Заметим, что имя файла должно быть полным либо должно строиться относительно той директории, которая является текущей для процесса, совершившего вызов. Если имя файла относится к файлу типа «связь», то читается информация (рекурсивно!) об атрибутах файла, на который указывает символическая связь.

Системный вызов **lstat** идентичен системному вызову **stat** за одним исключением: если имя файла относится к файлу типа «связь», то читается информация о самом файле типа «связь».

Системный вызов **fstat** идентичен системному вызову **stat**, только файл задается не именем, а своим файловым дескриптором (естественно, файл к этому моменту должен быть открыт).

Для системных вызовов **stat** и **lstat** процессу не нужны никакие права доступа к указанному файлу, но могут понадобиться права для поиска во всех директориях, входящих в специфицированное имя файла.

Структура **stat** в различных версиях **UNIX** может быть описана по-разному. В **Linux** она содержит следующие поля:

```
struct stat {  
    dev_t st_dev; /* устройство, на котором расположен файл */  
    ino_t st_ino; /* номер индексного узла для файла */  
    mode_t st_mode; /* тип файла и права доступа к нему */  
    nlink_t st_nlink; /* счетчик числа жестких связей */  
    uid_t st_uid; /* идентификатор пользователя владельца */  
    gid_t st_gid; /* идентификатор группы владельца */  
    dev_t st_rdev; /* тип устройства для специальных файлов  
    устройства */  
    off_t st_size; /* размер файла в байтах (если определен для данного  
    типа файлов) */  
    unsigned long st_blksize; /* размер блока для файловой системы */  
    unsigned long st_blocks; /* число выделенных блоков */  
    time_t st_atime; /* время последнего доступа к файлу */  
    time_t st_mtime; /* время последней модификации файла */  
    time_t st_ctime; /* время создания файла */  
}
```

Для определения типа файла можно использовать следующие логические макросы, применяя их к значению поля **st_mode**:

- **S_ISLNK(m)** – файл типа «связь».
- **S_ISREG(m)** – регулярный файл.
- **S_ISDIR(m)** – директория.
- **S_ISCHR(m)** – специальный файл символьного устройства.
- **S_ISBLK(m)** – специальный файл блочного устройства.
- **S_ISFIFO(m)** – файл типа FIFO.
- **S_ISSOCK(m)** – файл типа «socket».

Младшие 9 бит поля `st_mode` определяют права доступа к файлу подобно тому, как это делается в маске создания файлов текущего процесса.

Возвращаемое значение

Системные вызовы возвращают значение **0** при нормальном завершении и значение **-1** при возникновении ошибки.

Операции изменения атрибутов файла. Большинство операций изменения атрибутов файла обычно выполняется пользователем в интерактивном режиме с помощью команд операционной системы. Отметим операцию изменения размеров файла, а точнее операцию его обрезания без изменения всех других атрибутов. Для того чтобы уменьшить размеры существующего файла до **0**, не затрагивая остальных его характеристик (прав доступа, даты создания, учетной информации и т.д.), можно при открытии файла использовать в комбинации флагов системного вызова **open()** флаг **O_TRUNC**. Для изменения размеров файла до любой желаемой величины (даже для его увеличения во многих вариантах **UNIX**, хотя изначально этого не предусматривалось!) может использоваться системный вызов **ftruncate()**. При этом, если размер файла уменьшается, то вся информация в конце файла, не помещающаяся в новый размер, будет потеряна. Если же размер файла увеличивается, то это будет выглядеть так, как будто мы дополнили его до недостающего размера нулевыми байтами.

Системный вызов `ftruncate()`

Прототип системного вызова

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

```
#include <unistd.h>
```

```
int ftruncate(int fd, size_t length);
```

Описание системного вызова

Системный вызов **ftruncate** предназначен для изменения длины открытого регулярного файла.

Параметр **fd** является дескриптором соответствующего файла, т. е. значением, которое вернул системный вызов **open()**.

Параметр **length** – значение новой длины для этого файла. Если параметр **length** меньше, чем текущая длина файла, то вся информация в конце файла, не влезаящая в новый размер, будет потеряна. Если же он больше, чем текущая длина, то файл будет выглядеть так, как будто дополнили его до недостающего размера нулевыми байтами.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Операции чтения из файла и записи в файл. Для операций чтения из файла и записи в файл применяются системные вызовы **read()** и **write()**.

Необходимо отметить, что их поведение при работе с файлами имеет определенные особенности, связанные с понятием указателя текущей позиции в файле.

При работе с файлами информация записывается в файл или читается из него, начиная с места, определяемого указателем текущей позиции в файле. Значение указателя увеличивается на количество реально прочитанных или записанных байт. При чтении информации из файла она не пропадает из него. Если системный вызов **read** возвращает значение **0**, то это означает, что достигнут конец файла.

Операция изменения указателя текущей позиции.

Системный вызов **lseek()**. С точки зрения процесса все регулярные файлы являются файлами прямого доступа. В любой момент процесс может изменить положение указателя текущей позиции в открытом файле с помощью системного вызова **lseek()**.

Особенностью этого системного вызова является возможность помещения указателя текущей позиции в файле за конец файла (т. е. возможность установления значения указателя большего, чем длина файла).

При любой последующей операции записи в таком положении указателя файл будет выглядеть так, как будто возникший промежуток от конца файла до текущей позиции, где начинается запись, был заполнен нулевыми байтами. Если операции записи в таком положении указателя не производится, то никакого изменения файла,

связанного с необычным значением указателя, не произойдет (например, операция чтения будет возвращать нулевое значение для количества прочитанных байтов).

Системный вызов **lseek()**

Прототип системного вызова

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

```
#include <unistd.h>
```

```
off_t lseek(int fd, off_t offset, int whence);
```

Описание системного вызова

Системный вызов **lseek** предназначен для изменения положения указателя текущей позиции в открытом регулярном файле.

Параметр **fd** является дескриптором соответствующего файла, т. е. значением, которое вернул системный вызов **open()**.

Параметр **offset** совместно с параметром **whence** определяют новое положение указателя текущей позиции следующим образом:

- Если значение параметра **whence** равно **SEEK_SET**, то новое значение указателя будет составлять **offset** байт от начала файла. Естественно, что значение **offset** в этом случае должно быть не отрицательным.
- Если значение параметра **whence** равно **SEEK_CUR**, то новое значение указателя будет составлять старое значение указателя + **offset** байт. При этом новое значение указателя не должно стать отрицательным.
- Если значение параметра **whence** равно **SEEK_END**, то новое значение указателя будет составлять длина файла + **offset** байт. При этом новое значение указателя не должно стать отрицательным.

Системный вызов **lseek** позволяет выставить текущее значение указателя за конец файла (т.е. сделать его превышающим размер файла). При любой последующей операции записи в этом положении указателя файл будет выглядеть так, как будто возникший промежуток был заполнен нулевыми битами.

Тип данных **off_t** обычно является синонимом типа **long**.

Возвращаемое значение

Системный вызов возвращает новое положение указателя текущей позиции в байтах от начала файла при нормальном завершении и значение -1 при возникновении ошибки.

Операция добавления информации в файл. Флаг **O_APPEND**. Если открытие файла системным вызовом **open()** производилось с установленным флагом **O_APPEND**, то любая операция записи в файл будет всегда добавлять новые данные в конец файла, независимо от предыдущего положения указателя текущей позиции (как если бы непосредственно перед записью был выполнен вызов **lseek()** для установки указателя на конец файла).

Операции создания связей. Команда **ln**, системные вызовы **link()** и **symlink()**. Операции создания связи служат для проведения новых именованных ребер в уже существующей структуре без добавления новых узлов или для опосредованного проведения именованного ребра к уже существующему узлу через файл типа «связь» и неименованное ребро. Допустим, что несколько программистов совместно ведут работу над одним и тем же проектом. Файлы, относящиеся к этому проекту, вполне естественно могут быть выделены в отдельную директорию так, чтобы не смешиваться с файлами других пользователей и другими файлами программистов, участвующих в проекте. Для удобства каждый из разработчиков, конечно, хотел бы, чтобы эти файлы находились в его собственной директории. Этого можно было бы добиться, копируя по мере изменения новые версии соответствующих файлов из директории одного исполнителя в директорию другого исполнителя. Однако тогда, во-первых, возникнет ненужное дублирование информации на диске. Во-вторых, появится необходимость решения тяжелой задачи: синхронизации обновления замены всех копий этих файлов новыми версиями.

Существует другое решение проблемы. Достаточно разрешить файлам иметь несколько имен. Тогда одному физическому экземпляру данных на диске могут соответствовать различные имена файла, находящиеся в одной или в разных директориях. Подобная операция присвоения нового имени файлу (без уничтожения ранее существовавшего имени) получила название операции создания связи.

В операционной системе UNIX связь может быть создана двумя различными способами.

Первый способ, наиболее точно следующий описанной выше процедуре, получил название способа создания жесткой связи (hard link). С точки зрения логической структуры файловой системы этому способу соответствует проведение нового именованного ребра из узла, соответствующего некоторой директории, к узлу, соответствующему файлу любого типа, получающему дополнительное имя. С точки зрения структур данных, описывающих строение файловой системы, в эту директорию добавляется запись, содержащая дополнительное имя файла и номер его индексного узла (уже существующий!). При таком подходе и новое имя файла, и его старое имя или имена абсолютно равноправны для операционной системы и могут взаимозаменяемо использоваться для осуществления всех операций.

Использование жестких связей приводит к возникновению двух проблем.

Первая проблема связана с операцией удаления файла. Если необходимо удалить файл из некоторой директории, то после удаления из ее содержимого записи, соответствующей этому файлу, нельзя освободить логические блоки, занимаемые файлом, и его индексный узел, не убедившись, что у файла нет дополнительных имен (к его индексному узлу не ведут ссылки из других директорий), иначе нарушится целостность файловой системы. Для решения этой проблемы файлы получают дополнительный атрибут – счетчик жестких связей (или именованных ребер), ведущих к ним, который, как и другие атрибуты, располагается в их индексных узлах. При создании файла этот счетчик получает значение 1. При создании каждой новой жесткой связи, ведущей к файлу, он увеличивается на 1. При удалении файла из некоторой директории из ее содержимого удаляется запись об этом файле, и счетчик жестких связей уменьшается на 1. Если его значение становится равным 0, происходит освобождение логических блоков и индексного узла, выделенных этому файлу.

Вторая проблема связана с опасностью превращения логической структуры файловой системы из ациклического графа в циклический и с возможной неопределенностью толкования записи с именем "..." в содержимом директорий. Для их предотвращения во всех существующих вариантах операционной системы UNIX запрещено

создание жестких связей, ведущих к уже существующим директориям (несмотря на то, что POSIX-стандарт для операционной системы UNIX разрешает подобную операцию для пользователя root). В операционной системе Linux по непонятной причине дополнительно запрещено создание жестких связей, ведущих к специальным файлам устройств.

Команда **ln**

Синтаксис команды

ln [options] source [dest]

ln [options] source ... directory

Описание команды

Команда **ln** предназначена для реализации операции создания связи в файловой системе. В курсе будет использоваться две формы этой команды.

Первая форма команды, когда в качестве параметра **source** задается имя только одного файла, а параметр **dest** отсутствует. Или когда в качестве параметра **dest** задается имя файла, не существующего в файловой системе, создает связь к файлу, указанному в качестве параметра **source**, в текущей директории с его именем (если параметр **dest** отсутствует) или с именем **dest** (полным или относительным) в случае наличия параметра **dest**.

Вторая форма команды, когда в качестве параметра **source** задаются имена одного или нескольких файлов, разделенные между собой пробелами. А в качестве параметра **directory** задается имя уже существующей в файловой системе директории, создаются связи к каждому из файлов, перечисленных в параметре **source**, в директории **directory** с именами, совпадающими с именами перечисленных файлов.

Команда **ln** без опций служит для создания жестких связей (**hard link**), а команда **ln** с опцией **-s** – для создания мягких (**soft link**) или символических (**symbolic**) связей. Для создания жестких связей применяются команда операционной системы **ln** без опций и системный вызов **link()**.

Необходимо отметить, что системный вызов **link()** является одним из немногих системных вызовов, совершающих над файлами операции, которые не требуют предварительного открытия файла,

поскольку он подразумевает выполнение единичного действия только над содержимым индексного узла, выделенного связываемому файлу.

Системный вызов link()

Прототип системного вызова

#include <unistd.h>

**int link(char *pathname,
 char *linkpathname);**

Описание системного вызова.

Системный вызов **link** служит для создания жесткой связи к файлу с именем, на которое указывает параметр **pathname**. Указатель на имя создаваемой связи задается параметром **linkpathname** (полное или относительное имя связи). Во всех существующих реализациях операционной системы UNIX запрещено создавать жесткие связи к директориям. В операционной системе Linux (по непонятной причине) дополнительно запрещено создавать жесткие связи к специальным файлам устройств.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Жесткая связь файлов является аналогом использования прямых ссылок (указателей) в современных языках программирования, символическая или мягкая связь, до некоторой степени, напоминает косвенные ссылки (указатель на указатель). При создании мягкой связи с именем **symlink** из некоторой директории к файлу, заданному полным или относительным именем **linkpath**. В этой директории действительно создается новый файл типа «связь» с именем **symlink** со своими собственными индексным узлом и логическими блоками. При тщательном рассмотрении можно обнаружить, что все его содержимое составляет только символьная запись имени **linkpath**.

Операция открытия файла типа «связь» устроена таким образом, что в действительности открывается не сам этот файл, а тот файл, чье имя содержится в нем (при необходимости рекурсивно!). Поэтому операции над файлами, требующие предварительного

открытия файла, в реальности будут совершаться не над файлом типа «связь», а над тем файлом, имя которого содержится в нем (или над тем файлом, который, в конце концов, откроется при рекурсивных ссылках). Отсюда в частности следует, что попытки прочитать реальное содержимое файлов типа «связь» с помощью системного вызова **read()** обречены на неудачу. Как видно, создание мягкой связи, с точки зрения изменения логической структуры файловой системы, эквивалентно опосредованному проведению именованного ребра к уже существующему узлу через файл типа «связь» и неименованное ребро. Создание символической связи не приводит к проблеме, связанной с удалением файлов. Если файл, на который ссылается мягкая связь, удаляется с физического носителя, то попытка открытия файла мягкой связи (а, следовательно, и удаленного файла) приведет к ошибке «Файла с таким именем не существует», которая может быть аккуратно обработана приложением. Таким образом, удаление связанного объекта, как упоминалось ранее, лишь отчасти и не фатально нарушит целостность файловой системы.

Неаккуратное применение символических связей пользователями операционной системы может привести к превращению логической структуры файловой системы из ациклического графа в циклический граф. Это, конечно, нежелательно, но не носит столь разрушительного характера, как циклы, которые могли бы быть созданы жесткой связью, если бы не был введен запрет на организацию жестких связей к директориям. Поскольку мягкие связи принципиально отличаются от жестких связей и связей, возникающих между директорией и файлом при его создании, мягкая связь легко может быть идентифицирована операционной системой или программой пользователя. Для предотвращения заикливания программ, выполняющих операции над файлами, обычно ограничивается глубина рекурсии по прохождению мягких связей. Превышение этой глубины приводит к возникновению ошибки «Слишком много мягких связей», которая может быть легко обработана приложением. Поэтому ограничения на тип файлов, к которым может вести мягкая связь, в операционной системе UNIX не вводятся.

Для создания мягких связей применяются уже знакомая нам команда операционной системы **ln** с опцией **-s** и системный вызов **symlink()**. Надо отметить, что системный вызов **symlink()** также не

требует предварительного открытия связываемого файла, поскольку он вообще не рассматривает его содержимое.

Системный вызов `symlink()`

Прототип системного вызова

#include <unistd.h>

```
int symlink(char *pathname,  
            char *linkpathname);
```

Описание системного вызова

Системный вызов **symlink** служит для создания символической (мягкой) связи к файлу с именем, на которое указывает параметр **pathname**. Указатель на имя создаваемой связи задается параметром **linkpathname** (полное или относительное имя связи).

Никакой проверки реального существования файла с именем **pathname** системный вызов не производит.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Операция удаления связей и файлов. Системный вызов `unlink()`. При рассмотрении операции связывания файлов мы уже почти полностью рассмотрели, как производится операция удаления жестких связей и файлов. При удалении мягкой связи, т.е. фактически файла типа «связь», все происходит, как и для обычных файлов. Единственным изменением, с точки зрения логической структуры файловой системы, является то, что при действительном удалении узла, соответствующего файлу типа «связь», вместе с ним удаляется и выходящее из него неименованное ребро. Дополнительно необходимо отметить, что условием реального удаления регулярного файла с диска является не только равенство 0 значения его счетчика жестких связей, но и отсутствие процессов, которые держат этот файл открытым. Если такие процессы есть, то удаление регулярного файла будет выполнено при его полном закрытии последним использующим файл процессом. Для осуществления операции удаления жестких связей и/или файлов можно задействовать команду операционной системы `rm` или

системный вызов **unlink()**. Системный вызов **unlink()** также не требует предварительного открытия удаляемого файла, поскольку после его удаления совершать над ним операции бессмысленно.

Системный вызов **unlink()**

Прототип системного вызова

#include <unistd.h>

int unlink(char *pathname);

Описание системного вызова

Системный вызов **unlink** служит для удаления имени, на которое указывает параметр **pathname**, из файловой системы.

- Если после удаления имени счетчик числа жестких связей у данного файла стал равным 0, то возможны следующие ситуации.
- Если в операционной системе нет процессов, которые держат данный файл открытым, то файл полностью удаляется с физического носителя.
- Если удаляемое имя было последней жесткой связью для регулярного файла, но какой-либо процесс держит его открытым, то файл продолжает существовать до тех пор, пока не будет закрыт последний файловый дескриптор, ссылающийся на данный файл.
- Если имя относится к файлу типа socket, FIFO или к специальному файлу устройства, то файл удаляется независимо от наличия процессов, держащих его открытым, но процессы, открывшие данный объект, могут продолжать пользоваться им.
- Если имя относится к файлу типа «связь», то он удаляется, и мягкая связь оказывается разорванной.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

8.8 Специальные функции для работы с содержимым директорий

Стандартные системные вызовы **open()**, **read()** и **close()** не могут помочь программисту изучить содержимое файла типа «директория». Для анализа содержимого директорий используется набор функций из стандартной библиотеки языка Си.

Функция **opendir()**

Прототип функции

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

```
#include <dirent.h>
```

```
DIR *opendir(char *name);
```

Описание функции

Функция **opendir** служит для открытия потока информации для директории, имя которой расположено по указателю *name*. Тип данных **DIR** представляет собой некоторую структуру данных, описывающую такой поток. Функция **opendir** подготавливает почву для функционирования других функций, выполняющих операции над директорией, и позиционирует поток на первой записи директории.

Возвращаемое значение

При удачном завершении функция возвращает указатель на открытый поток директории, который будет в дальнейшем передаваться в качестве параметра всем другим функциям, работающим с этой директорией. При неудачном завершении возвращается значение **NULL**.

С точки зрения программиста в этом интерфейсе директория представляется как файл последовательного доступа, над которым можно совершать операции чтения очередной записи и позиционирования на начале файла. Перед выполнением этих операций директорию необходимо открыть, а после окончания – закрыть. Чтение очередной записи из директории осуществляет функция **readdir()**, одновременно позиционируя нас на начале следующей записи (если она, конечно, существует). Для операции нового позиционирования на начале директории (если вдруг

понадобится) применяется функция `rewinddir()`. После окончания работы с директорией ее необходимо закрыть с помощью функции `closedir()`.

Функция `readdir()`

Прототип функции

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dir);
```

Описание функции

Функция **`readdir`** служит для чтения очередной записи из потока информации для директории.

Параметр **`dir`** представляет собой указатель на структуру, описывающую поток директории, который вернула функция **`opendir()`**. Тип данных **`struct dirent`** представляет собой некоторую структуру данных, описывающую одну запись в директории. Поля этой записи сильно варьируются от одной файловой системы к другой, но одно из полей, которое собственно и будет нас интересовать, всегда присутствует в ней. Это поле **`char d_name[]`** неопределенной длины, не превышающей значения **`NAME_MAX+1`**, которое содержит символьное имя файла, завершающееся символом конца строки. Данные, возвращаемые функцией **`readdir`**, переписываются при очередном вызове этой функции для того же самого потока директории.

Возвращаемое значение

При удачном завершении функция возвращает указатель на структуру, содержащую очередную запись директории. При неудачном завершении или при достижении конца директории возвращается значение `NULL`.

Функция `rewinddir()`

Прототип функции

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

void rewinddir(DIR *dir);

Описание функции

Функция **rewinddir** служит для позиционирования потока информации для директории, ассоциированного с указателем **dir** (т.е. с тем, что вернула функция **opendir()**), на первой записи (или на начале) директории.

Функция closedir()

Прототип функции

#include <sys/types.h>

#include <dirent.h>

int closedir(DIR *dir);

Описание функции

Функция **closedir** служит для закрытия потока информации для директории, ассоциированного с указателем **dir** (т.е. с тем, что вернула функция **opendir()**). После закрытия поток директории становится недоступным для дальнейшего использования.

Возвращаемое значение

При успешном завершении функция возвращает значение 0, при неудачном завершении – значение -1.

8.9 Понятие о файлах, отображаемых в память (memory mapped файлах). Системные вызовы mmap(), munmap()

С помощью системного вызова **open()** операционная система отображает файл из пространства имен в дисковое пространство файловой системы, подготавливая почву для осуществления других операций. С появлением концепции виртуальной памяти, когда физические размеры памяти перестали играть роль сдерживающего фактора в развитии вычислительных систем, стало возможным отображать файлы непосредственно в адресное пространство процессов. Иными словами, появилась возможность работать с файлами как с обычной памятью, заменив выполнение базовых операций над ними с помощью системных вызовов на использование

операций обычных языков программирования. Файлы, чье содержимое отображается непосредственно в адресное пространство процессов, получили название файлов, отображаемых в память (по-английски – *memory mapped* файлов). Необходимо отметить, что такое отображение может быть осуществлено не только для всего файла в целом, но и для его части. С точки зрения программиста работа с такими файлами выглядит следующим образом: отображение файла из пространства имен в адресное пространство процесса происходит в два этапа: сначала выполняется отображение в дисковое пространство, а уже затем из дискового пространства в адресное. Поэтому вначале файл необходимо открыть, используя обычный системный вызов **open()**. Вторым этапом является отображение файла целиком или частично из дискового пространства в адресное пространство процесса. Для этого используется системный вызов **mmap()**. Файл после этого можно, и закрыть, выполнив системный вызов **close()**, так как необходимую информацию о расположении файла на диске мы уже сохранили в других структурах данных при вызове **mmap()**.

Системный вызов **mmap()**

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/mman.h>
void *mmap (void *start, size_t length,
            int prot, int flags, int fd,
            off_t offset);
```

Описание системного вызова. Системный вызов **mmap** служит для отображения предварительно открытого файла (например, с помощью системного вызова **open()**) в адресное пространство вычислительной системы. После его выполнения файл может быть закрыт (например, системным вызовом **close()**), что никак не повлияет на дальнейшую работу с отображенным файлом. Настоящее описание не является полным описанием системного вызова. Для получения полной информации обращайтесь к **UNIX Manual**. Параметр **fd** является файловым дескриптором для файла, который нужно отобразить в адресное пространство (т. е. значением, которое вернул системный вызов **open()**).

Ненулевое значение параметра **start** может использоваться только очень квалифицированными системными программистами, поэтому в семинарах будем всегда полагать его равным значению **NULL**, позволяя операционной системе самой выбрать начало области адресного пространства, в которую будет отображен файл. В память будет отображаться часть файла, начиная с позиции внутри него, заданной значением параметра **offset** – смещение от начала файла в байтах, и длиной, равной значению параметра **length** (естественно, тоже в байтах). Значение параметра **length** может и превышать реальную длину от позиции **offset** до конца существующего файла. На поведении системного вызова это никак не отразится, но в дальнейшем при попытке доступа к ячейкам памяти, лежащим вне границ реального файла, возникнет сигнал **SIGBUS** (реакция на него по умолчанию – прекращение процесса с образованием **core** файла). Параметр **flags** определяет способ отображения файла в адресное пространство. В рамках курса будем использовать только два его возможных значения: **MAP_SHARED** и **MAP_PRIVATE**. Если в качестве его значения выбрано **MAP_SHARED**, то полученное отображение файла впоследствии будет использоваться и другими процессами, вызвавшими **mmap** для этого файла с аналогичными значениями параметров, а все изменения, сделанные в отображенном файле, будут сохранены во вторичной памяти. Если в качестве значения параметра **flags** указано **MAP_PRIVATE**, то процесс получает отображение файла в свое монопольное распоряжение, но все изменения в нем не могут быть занесены во вторичную память (т. е., проще говоря, не сохраняются). Параметр **prot** определяет разрешенные операции над областью памяти, в которую будет отображен файл. В качестве его значения мы будем использовать значения **PROT_READ** (разрешено чтение), **PROT_WRITE** (разрешена запись) или их комбинацию через операцию "побитовое или" – "**|**". Необходимо отметить две существенные особенности системного вызова, связанные с этим параметром:

Значение параметра **prot** не может быть шире, чем операции над файлом, заявленные при его открытии в параметре **flags** системного вызова **open()**. Например, нельзя открыть файл только для чтения, а при его отображении в память использовать значение **prot = PROT_READ | PROT_WRITE**.

В результате ошибки в операционной системе Linux при работе на 486-х и 586-х процессорах попытка записать в отображение файла, открытое только для записи, более 32 байт одновременно приводит к ошибке (возникает сигнал о нарушении защиты памяти).

Возвращаемое значение

При нормальном завершении системный вызов возвращает начальный адрес области памяти, в которую отображен файл (или его часть), при возникновении ошибки – специальное значение **MAP_FAILED**.

После этого с содержимым файла можно работать, как с содержимым обычной области памяти.

По окончании работы с содержимым файла, необходимо освободить дополнительно выделенную процессу область памяти, предварительно синхронизировав содержимое файла на диске с содержимым этой области (если, конечно, необходимо). Эти действия выполняет системный вызов **munmap()**.

Системный вызов munmap

Прототип системного вызова

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

```
#include <unistd.h>
```

```
#include <sys/mman.h>
```

```
int munmap (void *start, size_t length);
```

Описание системного вызова. Системный вызов **munmap** служит для прекращения отображения методом **mmap** файла в адресное пространство вычислительной системы. Если при системном вызове **mmap()** было задано значение параметра **flags**, равное **MAP_SHARED**, и в отображении файла была разрешена операция записи (в параметре **prot** использовалось значение **PROT_WRITE**), то **munmap** синхронизирует содержимое отображения с содержимым файла во вторичной памяти. После его выполнения области памяти, использовавшиеся для отображения файла, становятся недоступны текущему процессу. Параметр **start** является адресом начала области памяти, выделенной для отображения файла, т. е. значением, которое вернул системный вызов **mmap()**. Параметр **length** определяет ее

длину, и его значение должно совпадать со значением соответствующего параметра в системном вызове **mmap()**.

Возвращаемое значение

При нормальном завершении системный вызов возвращает значение 0, при возникновении ошибки – значение -1.

Программа 8-01 для иллюстрации работы с memory mapped файлом

```
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/mman.h>
#include <iostream>
#include <stdio.h>
using namespace std;
int main(void)
{
    int fd; /* Файловый дескриптор для файла, в котором будет
храниться информация*/
    size_t length; /* Длина отображаемой части файла */
    int i;
    /* Ниже следует описание типа структуры, которым мы заведем
файл, и двух указателей на подобный тип. Указатель ptr будет
использоваться в качестве начального адреса выделенной области
памяти, а указатель tmp_ptr – для перемещения внутри этой области.
*/
    struct A
    {
        double f;
        double f2;
    }
    *ptr, tmp_ptr;

    /* Открытие файла или сначала создание его (если такого файла
не было). Права доступа к файлу при создании определяются как
read-write для всех категорий пользователей (0666). Из-за ошибки в
```


Linux необходимо ниже в системном вызове `mmap()` разрешить в отображении файла и чтение, и запись, хотя реально нужна только запись. Поэтому и при открытии файла необходимо задавать `O_RDWR`. */

```
fd = open("mapped.dat", O_RDWR | O_CREAT, 0666);
if( fd == -1)
{
    /* Если файл открыть не удалось, вывод сообщения об ошибке и
    завершение работы */
    cout<<"File open failed!"<<endl;
    exit(1);
}
/* Вычисление будущей длины файла (для 100000 структур) */
length = 100000*sizeof(struct A);
/* Вновь созданный файл имеет длину 0. Если его отобразить в
память с такой длиной, то любая попытка записи в выделенную
память приведет к ошибке. Увеличение длины файла с помощью
вызова ftruncate(). */
ftruncate(fd,length);
/* Отображаем файл в память. Разрешенные операции над
отображением указываем как PROT_WRITE | PROT_READ по уже
названным причинам. Значение флагов ставится в MAP_SHARED, так
как необходимо сохранить информацию, которая занесена в
отображение, на диске. Файл отображаем с его начала (offset = 0) и
до конца (length = длине файла). */
ptr = (struct A )mmap(NULL, length, PROT_WRITE | PROT_READ,
MAP_SHARED, fd, 0);
/* Файловый дескриптор более не нужен, и его можно закрыть */
close(fd);
if( ptr == MAP_FAILED )
{
    /* Если отобразить файл не удалось, сообщение об ошибке и
    завершение работы */
    cout<<"Mapping failed!"<< endl;
    exit(2);
}
```

```

/* В цикле заполнение образа файла числами от 1 до 100000 и их
квадратами. Для перемещения по области памяти используется
указатель tmpptr, так как указатель ptr на начало образа файла
понадобится для прекращения и отображения вызовом типтар(). */
tmpptr = ptr;
for(i = 1; i <= 100000; i++)
{
    tmpptr->f = i;
    tmpptr->f2 = tmpptr->f*tmpptr->f;
    tmpptr++;
}
/* Прекращение отображения файла в память, запись
содержимого отображения на диск и освобождение памяти. */
типтар((void *)ptr, length);
return 0;
}

```

Листинг 8-01. Программа для иллюстрации работы с memory mapped файлом.

Эта программа создает файл, отображает его в адресное пространство процесса и заносит в него информацию с помощью обычных операций языка Си. Обратите внимание на необходимость увеличения размера файла перед его отображением. Созданный файл имеет нулевой размер, и, если его с этим размером отобразить в память, можно записать в него или прочитать из него не более 0 байт, т. е. ничего. Для увеличения размера файла использован системный вызов **ftruncate()**, хотя это можно было бы сделать и любым другим способом. При отображении файла необходимо разрешить в нем и запись, и чтение, хотя реально происходит только запись. Это сделано для того, чтобы избежать ошибки в операционной системе Linux, связанной с использованием 486-х и 586-х процессоров. Такой список разрешенных операций однозначно требует, чтобы при открытии файла системным вызовом **open()** файл открывался и на запись, и на чтение. Поскольку информацию мы желаем сохранить на диске, при отображении использовано значение флагов **MAP_SHARED**.

Порядок выполнения лабораторной работы

1. Напишите программу для перемещения файлов из одной директории в другую, у которых в названии содержится определенное слово. Первый аргумент - исходная директория. Второй аргумент - директория, в которую необходимо переместить файлы. Третий аргумент - ключевое слово, которое должно содержаться в названии файлов. Откомпилируйте, запустите
2. Модифицируйте программу 7-01 так, чтобы она отображала файл, записанный программой из раздела «Анализ, компиляция и прогон программы для создания метогу mapped файла и записи его содержимого», в память и считала сумму квадратов чисел от 1 до 100000, которые уже находятся в этом файле.
3. Напишите, откомпилируйте и прогоните программу, распечатывающую список файлов, входящих в директорию, с указанием их типов. Имя директории задается как параметр командной строки. Если оно отсутствует, то выбирается текущая директория.
4. Создайте жесткие и символические связи из вашей директории к другим файлам. Просмотрите содержимое директорий со связями с помощью команды `ls -al`. Обратите внимание на отличие мягких и жестких связей в листинге этой команды. Определите допустимую глубину рекурсии символических связей для вашей операционной системы.
5. Напишите программу, распечатывающую содержимое заданной директории в формате, аналогичном формату выдачи команды `ls -al`. Для этого вам дополнительно понадобится самостоятельно изучить в UNIX Manual функцию `ctime(3)` и системные вызовы `time(2)`, `readlink(2)`. Цифры после имен функций и системных вызовов – это номера соответствующих разделов для UNIX Manual.
6. Напишите две программы, использующие метогу mapped файл для обмена информацией при одновременной работе подобно тому, как они могли бы использовать разделяемую память

Контрольные вопросы

1. Понятие виртуальной файловой системы.
2. Опишите известные вам операции над файловыми системами. Монтирование файловых систем.
3. Блочные, символьные устройства. Понятие драйвера. Блочные, символьные драйверы, драйверы низкого уровня. Файловый интерфейс.
4. Аппаратные прерывания (interrupt), исключения (exception), программные прерывания (trap, software interrupt). Их обработка.
5. Понятие сигнала. Способы возникновения сигналов и виды их обработки.
6. Понятия группы процессов, сеанса, лидера группы, лидера сеанса, управляющего терминала сеанса. Системные вызовы `getpgrp()`, `setpgrp()`, `getpgid()`, `setpgid()`, `getsid()`, `setsid()`.
7. Системный вызов `kill()` и команда `kill`.
8. Изучение особенностей получения терминальных сигналов текущей и фоновой группой процессов.
9. Системный вызов `signal()`. Установка собственного обработчика сигнала.
10. Поясните листинг и реализацию программы, игнорирующей сигнал `SIGINT`.
11. Поясните листинг и реализацию программы с пользовательской обработкой сигнала `SIGINT`.
12. Сигналы `SIGUSR1` и `SIGUSR2`. Использование сигналов для синхронизации процессов.
13. Завершение порожденного процесса. Системный вызов `waitpid()`. Сигнал `SIGCHLD`.
14. Прогон программы для иллюстрации обработки сигнала `SIGCHLD`.
15. Возникновение сигнала `SIGPIPE` при попытке записи в `pipe` или `FIFO`, который никто не собирается читать.
- 16 Понятие о надежности сигналов. POSIX функции для работы с сигналами

Лабораторная работа 9

ОРГАНИЗАЦИЯ ВВОДА-ВЫВОДА В UNIX. ФАЙЛЫ УСТРОЙСТВ. АППАРАТ ПРЕРЫВАНИЙ. СИГНАЛЫ В UNIX

Цели и задачи

Рассмотреть организацию ввода-вывода в UNIX. Рассмотреть возможные файлы устройств и аппарат прерываний. Изучить реализацию сигналов в UNIX.

9.1 Понятие виртуальной файловой системы

Современные версии UNIX-подобных операционных систем умеют работать с разнообразными файловыми системами, различающимися своей организацией. Такая возможность достигается с помощью разбиения каждой файловой системы на зависимую и независимую от конкретной реализации части. Независимые части всех файловых систем одинаковы и представляют для всех остальных элементов ядра абстрактную файловую систему, которую принято называть виртуальной файловой системой. Зависимые части для различных файловых систем могут встраиваться в ядро на этапе компиляции, либо добавляться к нему динамически по мере необходимости, без перекомпиляции системы (как в системах с микроядерной архитектурой). В файловой системе **s5fs** данные о физическом расположении и атрибутах каждого открытого файла представляются в операционной системе структурой данных в таблице индексных узлов открытых файлов, содержащей информацию из индексного узла файла во вторичной памяти. В виртуальной файловой системе, в отличие от **s5fs**, каждый файл характеризуется не индексным узлом **inode**, а некоторым виртуальным узлом **vnode**. Соответственно, вместо таблицы индексных узлов открытых файлов в операционной системе появляется таблица виртуальных узлов открытых файлов. При открытии файла в операционной системе для него заполняется (если не был заполнен раньше) элемент таблицы виртуальных узлов открытых файлов. В нём хранятся, как минимум, тип файла, счетчик числа открытий файла, указатель на реальные физические данные файла и, обязательно, указатель на таблицу системных вызовов, совершающих операции над файлом, – таблицу

операций. Реальные физические данные файла (равно как и способ расположения файла на диске и т.п.) и системные вызовы, реально выполняющие операции над файлом, уже не являются элементами виртуальной файловой системы. Они относятся к одной из зависимых частей файловой системы, так как определяются ее конкретной реализацией.

При выполнении операций над файлами по таблице операций, чей адрес содержится в **vnode**, определяется системный вызов, который будет на самом деле выполнен над реальными физическими данными файла, чей адрес также находится в **vnode**. Таблица операций является общей для всех файлов, принадлежащих одной и той же файловой системе.

9.2 Операции над файловыми системами. Монтирование файловых систем

В предыдущих разделах рассматривалась только одна файловая система, расположенная в одном разделе физического носителя. Как только предполагаем сосуществование нескольких файловых систем в рамках одной операционной системы, то встает вопрос о логическом объединении структур этих файловых систем. При работе операционной системы изначально доступна лишь одна, так называемая корневая, файловая система. Прежде, чем приступить к работе с файлом, лежащим в некоторой другой файловой системе, мы должны встроить ее в уже существующий ациклический граф файлов. Эта операция над файловой системой называется монтированием файловой системы (**mount**). Для монтирования файловой системы в существующем графе должна быть найдена или создана некоторая пустая директория – точка монтирования, к которой и присоединится корень монтируемой файловой системы. При операции монтирования в ядре заводятся структуры данных, описывающие файловую систему, а в **vnode** для точки монтирования файловой системы помещается специальная информация. Монтирование файловых систем обычно является прерогативой системного администратора и осуществляется командой операционной системы **mount** в ручном режиме, либо автоматически при старте операционной системы. Использование этой команды без параметров не требует специальных полномочий и позволяет

пользователю получить информацию обо всех смонтированных файловых системах и соответствующих им физических устройствах. Для пользователя также обычно разрешается монтирование файловых систем, расположенных на гибких магнитных дисках. Для первого накопителя на гибких магнитных дисках такая команда в Linux будет выглядеть следующим образом:

mount /dev/fd0 <имя пустой директории>,

где <имя пустой директории> описывает точку монтирования, а /dev/fd0 – специальный файл устройства, соответствующего этому накопителю.

Команда **mount**

Синтаксис команды

mount [-hV]

mount [-rw] [-t fstype] device dir

Описание команды

Команда **mount** предназначена для выполнения операции монтирования файловой системы и получения информации об уже смонтированных файловых системах.

Опции **-h**, **-V** используются при вызове команды без параметров и служат для следующих целей:

-h – вывести краткую инструкцию по пользованию командой;

-V – вывести информацию о версии команды **mount**;

Команда **mount** без опций и без параметров выводит информацию обо всех уже смонтированных файловых системах.

Команда **mount** с параметрами служит для выполнения операции монтирования файловой системы.

Параметр **device** задает имя специального файла для устройства, содержащего файловую систему.

Параметр **dir** задает имя точки монтирования (имя некоторой уже существующей пустой директории). При монтировании могут использоваться следующие опции:

-r — смонтировать файловую систему только для чтения (read only);

-w — смонтировать файловую систему для чтения и для записи (read/write).

-t fstype — задать тип монтируемой файловой системы как **fstype**.

Поддерживаемые типы файловых систем в операционной

системе Linux: adfs, affs, autofs, coda, coherent, cramfs, devpts, efs, ext, ext2, ext3, hfs, hpfs, iso9660 (для CD), minix, msdos, ncpfs, nfs, ntfs, proc, qnx4, reiserfs, romfs, smbfs, sysv, udf, ufs, umsdos, vfat, xenix, xfs, xiafs. При отсутствии явно заданного типа команда для большинства типов файловых систем способна опознать его автоматически. Если не собираемся использовать смонтированную файловую систему в дальнейшем (например, хотим вынуть ранее смонтированный диск), то необходимо выполнить операцию логического разъединения смонтированных файловых систем (**umount**). Для этой операции, которая тоже, как правило, является привилегией системного администратора, используется команда **umount** (может выполняться в ручном режиме или автоматически при завершении работы операционной системы).

Для пользователя обычно доступна команда отмонтирования файловой системы на диске в форме

umount <имя точки монтирования>

где <имя точки монтирования> – это <имя пустой директории>, использованное ранее в команде **mount**, или в форме **umount /dev/fd0**, где **/dev/fd0** – специальный файл устройства, соответствующего первому накопителю.

Заметим, что для последующей корректной работы операционной системы при удалении физического носителя информации необходимо предварительное логическое разъединение файловых систем, если они перед этим были объединены.

Команда **umount**

Синтаксис команды

umount [-hV]

umount device

umount dir

Описание команды

Команда **umount** предназначена для выполнения операции

логического разъединения ранее смонтированных файловых систем. Опции **-h**, **-V** используются при вызове команды без параметров и служат для следующих целей:

- h** – вывести краткую инструкцию по пользованию командой;
- V** – вывести информацию о версии команды `umount`.

Команда **umount** с параметром служит для выполнения операции логического разъединения файловых систем. В качестве параметра может быть задано либо имя устройства, содержащего файловую систему – **device**, либо имя точки монтирования файловой системы (т.е. имя директории, которое указывалось в качестве параметра при вызове команды `mount`) – **dir**.

Файловая система не может быть отмонтирована до тех пор, пока она находится в использовании (*busy*) – например, когда в ней существуют открытые файлы, какой-либо процесс имеет в качестве рабочей директории директорию в этой файловой системе и т. д.

9.3 Блочные, символьные устройства. Понятие драйвера. Блочные, символьные драйверы, драйверы низкого уровня. Файловый интерфейс

Все устройства ввода-вывода можно разделить на относительно небольшое число типов в зависимости от набора операций, которые могут ими выполняться. Такое деление позволяет организовать "слоистую" структуру подсистемы ввода-вывода, вынеся все аппаратно-зависимые части в драйверы устройств, с которыми взаимодействует базовая подсистема ввода-вывода, осуществляющая стратегическое управление всеми устройствами.

В операционной системе UNIX принята упрощенная классификация: все устройства разделяются по способу передачи данных на символьные и блочные. Символьные устройства осуществляют передачу данных байт за байтом, в то время как блочные устройства передают блок байт как единое целое. Типичным примером символьного устройства является клавиатура, примером блочного устройства – жесткий диск. Непосредственное взаимодействие операционной системы с устройствами ввода-вывода обеспечивают их драйверы. Существует пять основных случаев, когда ядро обращается к драйверам:

- **Автоконфигурация.** Происходит в процессе инициализации операционной системы, когда ядро определяет наличие доступных устройств.
- **Ввод-вывод.** Обработка запроса ввода-вывода.
- **Обработка прерываний.** Ядро вызывает специальные функции драйвера для обработки прерывания, поступившего от устройства, в том числе, возможно, для планирования очередности запросов к нему.
- **Специальные запросы.** Например, изменение параметров драйвера или устройства.
- **Повторная инициализация устройства или останов операционной системы.**

Как и устройства подразделяются на символьные и блочные, так и драйверы существуют символьные и блочные. Особенностью блочных устройств является возможность организации на них файловой системы, поэтому блочные драйверы обычно используются файловой системой UNIX. При обращении к блочному устройству, не содержащему файловой системы, применяются специальные драйверы низкого уровня, как правило, представляющие собой интерфейс между ядром операционной системы и блочным драйвером устройства. Для каждого из этих трех типов драйверов были выделены основные функции, которые базовая подсистема ввода-вывода может совершать над устройствами и драйверами: инициализация устройства или драйвера, временное завершение работы устройства, чтение, запись, обработка прерывания, опрос устройства и т. д. Эти функции были систематизированы и представляют собой интерфейс между драйверами и базовой подсистемой ввода-вывода. Каждый драйвер определенного типа в операционной системе UNIX получает собственный номер, который по сути дела является индексом в массиве специальных структур данных операционной системы – коммутаторе устройств соответствующего типа. Этот индекс принято также называть старшим номером устройства, хотя на самом деле он относится не к устройству, а к драйверу. Несмотря на наличие трех типов драйверов, в операционной системе используется всего два коммутатора: для блочных и символьных драйверов. Драйверы низкого уровня распределяются между ними по преобладающему типу

интерфейса (к какому типу ближе – в такой массив и заносится).

Каждый элемент коммутатора устройств обязательно содержит адреса (точки входа в драйвер), соответствующие стандартному набору функций интерфейса, которые и вызываются операционной системой для выполнения тех или иных действий над устройством и/или драйвером.

Помимо старшего номера устройства существует еще и младший номер устройства, который передается драйверу в качестве параметра и смысл которого определяется самим драйвером. Например, это может быть номер раздела на жестком диске (partition), доступ к которому должен обеспечить драйвер (надо отметить, что в операционной системе UNIX различные разделы физического носителя информации рассматриваются как различные устройства). В некоторых случаях младший номер устройства может не использоваться, но для единообразия он должен присутствовать. Таким образом, пара драйвер-устройство всегда однозначно определяется в операционной системе заданием пары номеров (старшего и младшего номеров устройства) и типа драйвера (символьный или блочный). Для связи приложений с драйверами устройств операционная система UNIX использует файловый интерфейс. Каждой тройке тип-драйвер-устройство в файловой системе соответствует специальный файл устройства, который не занимает на диске никаких логических блоков, кроме индексного узла. В качестве атрибутов этого файла помимо обычных атрибутов используются соответствующие старший и младший номера устройства и тип драйвера (тип драйвера определяется по типу файла: ибо есть специальные файлы символьных устройств и специальные файлы блочных устройств, а номера устройств занимают место длины файла, скажем, для регулярных файлов). Когда открывается специальный файл устройства, операционная система, в числе прочих действий, заносит в соответствующий элемент таблицы открытых виртуальных узлов указатель на набор функций интерфейса из соответствующего элемента коммутатора устройств. При попытке чтения из файла устройства или записи в файл устройства виртуальная файловая система будет транслировать запросы на выполнение этих операций в соответствующие вызовы нужного драйвера.

9.4 Аппаратные прерывания (interrupt), исключения (exception), программные прерывания (trap, software interrupt) и их обработка

После выдачи запроса ввода-вывода у процессора существует два способа узнать о том, что обработка запроса устройством завершена. Первый способ заключается в регулярной проверке процессором бита занятости в регистре состояния контроллера соответствующего устройства (polling). Второй способ заключается в использовании прерываний. При втором способе процессор имеет специальный вход, на который устройства ввода-вывода, используя контроллер прерываний или непосредственно, выставляют сигнал запроса прерывания (interrupt request) при завершении операции ввода-вывода. При наличии такого сигнала процессор после выполнения текущей команды не выполняет следующую, а, сохранив состояние ряда регистров и, возможно, загрузив в часть регистров новые значения, переходит к выполнению команд, расположенных по некоторым фиксированным адресам. После окончания обработки прерывания можно восстановить состояние процессора и продолжить его работу с команды, выполнение которой было отложено. Аналогичный механизм часто используется при обработке исключительных ситуаций (exception), возникающих при выполнении команды процессором (неправильный адрес в команде, защита памяти, деление на ноль и т.д.). В этом случае процессор не завершает выполнение команды, а поступает, как и при прерывании, сохраняя свое состояние до момента начала ее выполнения. Этим же механизмом часто пользуются и для реализации так называемых программных прерываний (software interrupt, trap), применяемых, например, для переключения процессора из режима пользователя в режим ядра внутри системных вызовов. Для выполнения действий, аналогичных действиям по обработке прерывания, процессор в этом случае должен выполнить специальную команду. Как правило, обработку аппаратных прерываний от устройств ввода-вывода производит сама операционная система, не доверяя работу с системными ресурсами процессам пользователя. Обработка же исключительных ситуаций и некоторых программных прерываний вполне может быть возложена на пользовательский процесс через механизм сигналов.

9.5 Понятие сигнала. Способы возникновения сигналов и виды их обработки

С точки зрения пользователя получение процессом сигнала выглядит как возникновение прерывания. Процесс прекращает регулярное исполнение, и управление передается механизму обработки сигнала. По окончании обработки сигнала процесс может возобновить регулярное исполнение. Типы сигналов (их принято задавать номерами, как правило, в диапазоне от 1 до 31 включительно или специальными символьными обозначениями) и способы их возникновения в системе жестко регламентированы.

Процесс может получить сигнал от:

- hardware (при возникновении исключительной ситуации);
- другого процесса, выполнившего системный вызов передачи сигнала;
- операционной системы (при наступлении некоторых событий);
- терминала (при нажатии определенной комбинации клавиш);
- системы управления заданиями (при выполнении команды kill).

Передачу сигналов процессу в случаях его генерации источниками 2, 3 и 5, т. е., в конечном счете, каким-либо другим процессом, можно рассматривать как реализацию в UNIX сигнальных средств связи.

Существует три варианта реакции процесса на сигнал:

- Принудительно проигнорировать сигнал.
- Произвести обработку по умолчанию: проигнорировать, остановить процесс (перевести в состояние ожидания до получения другого специального сигнала) либо завершить работу с образованием core файла или без него.
- Выполнить обработку сигнала, специфицированную пользователем.

Изменить реакцию процесса на сигнал можно с помощью специальных системных вызовов. Реакция на некоторые сигналы не допускает изменения, и они могут быть обработаны только по умолчанию. Так, например, сигнал с номером **9** – **SIGKILL** обрабатывается только по умолчанию и всегда приводит к завершению процесса.

Важным вопросом при программировании с использованием сигналов является вопрос о сохранении реакции на них при порождении нового процесса или замене его пользовательского контекста. При системном вызове **fork()** все установленные реакции на сигналы наследуются порожденным процессом.

При системном вызове **exec()** сохраняются реакции только для тех сигналов, которые игнорировались или обрабатывались по умолчанию. Получение любого сигнала, который до вызова **exec()** обрабатывался пользователем, приведет к завершению процесса.

9.6 Понятия группы процессов, сеанса, лидера группы, лидера сеанса, управляющего терминала сеанса.

Системные вызовы `getpgrp()`, `setpgrp()`, `getpgid()`, `setpgid()`, `getsid()`, `setsid()`

Все процессы в системе связаны родственными отношениями и образуют генеалогическое дерево или лес из таких деревьев, где в качестве узлов деревьев выступают сами процессы, а связями служат отношения родитель-ребенок. Все эти деревья принято разделять на группы процессов или семьи (рис. 9.1).

Группа процессов включает в себя один или более процессов и существует, пока в группе присутствует хотя бы один процесс. Каждый процесс обязательно включен в какую-нибудь группу. При рождении новый процесс попадает в ту же группу процессов, в которой находится его родитель. Процессы могут мигрировать из группы в группу по своему желанию или по желанию другого процесса (в зависимости от версии UNIX). Многие системные вызовы могут быть применены не к одному конкретному процессу, а ко всем процессам в некоторой группе. Поэтому то, как именно следует объединять процессы в группы, зависит от того, как предполагается их использовать. В свою очередь, группы процессов объединяются в сеансы, образуя, с родственной точки зрения, некие кланы семей. Понятие сеанса изначально было введено в UNIX для логического объединения групп процессов, созданных в результате каждого входа и последующей работы пользователя в системе. Поэтому с каждым сеансом может быть связан в системе терминал, называемый управляющим терминалом сеанса, через который обычно и общаются процессы сеанса с пользователем.

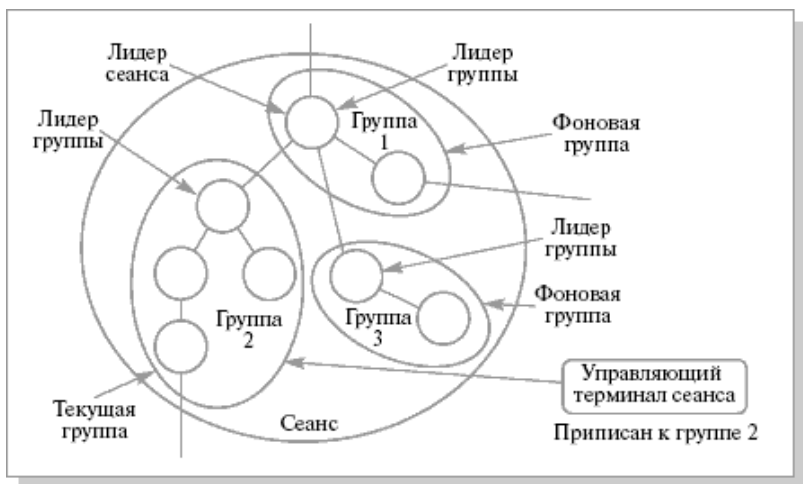


Рис. 9.1. Иерархия процессов в UNIX

Сеанс не может иметь более одного управляющего терминала, и один терминал не может быть управляющим для нескольких сеансов. В то же время могут существовать сеансы, вообще не имеющие управляющего терминала.

Каждая группа процессов в системе получает собственный уникальный номер. Узнать этот номер можно с помощью системного вызова `getpgid()`. Используя его, процесс может узнать номер группы для себя самого или для процесса из своего сеанса. Не во всех версиях UNIX присутствует данный системный вызов. Здесь возникает столкновение с тяжелым наследием разделения линий UNIX'ов на линию BSD и линию System V. Вместо вызова `getpgid()` в таких системах существует системный вызов `getpgrp()`, который возвращает номер группы только для текущего процесса.

Системный вызов `getpgid()`

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpgid(pid_t pid);
```

Описание системного вызова

Системный вызов возвращает идентификатор группы процессов для процесса с идентификатором **pid**. Узнать номер группы процесс может только для себя самого или для процесса из своего сеанса. При других значениях **pid** системный вызов возвращает значение **-1**.

Тип данных **pid_t** является синонимом для одного из целочисленных типов языка Си.

Системный вызов **getpgrp()**

Прототип системного вызова

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

```
#include <unistd.h>
```

```
pid_t getpgrp(void);
```

Описание системного вызова

Системный вызов **getpgrp** возвращает идентификатор группы процессов для текущего процесса.

Для перевода процесса в другую группу процессов, возможно, с одновременным ее созданием, применяется системный вызов **setpgid()**. Перевести в другую группу процесс может либо сам себя (и то не во всякую и не всегда), либо свой процесс-ребенок, который не выполнял системный вызов **exec()**, т. е. не запускал на выполнение другую программу. При определенных значениях параметров системного вызова создается новая группа процессов с идентификатором, совпадающим с идентификатором переводимого процесса, состоящая первоначально только из одного этого процесса. Новая группа может быть создана только таким способом, поэтому идентификаторы групп в системе уникальны. Переход в другую группу без создания новой группы возможен лишь в пределах одного сеанса. В некоторых разновидностях UNIX системный вызов **setpgid()** отсутствует, а вместо него используется системный вызов **setpgrp()**, способный только создавать новую группу процессов с идентификатором, совпадающим с идентификатором текущего процесса, и переводить в нее текущий процесс. (В ряде систем, где сосуществуют вызовы **setpgrp()** и **setpgid()**, например в Solaris, вызов **setpgrp()** ведет себя иначе – он аналогичен рассматриваемому ниже вызову **setsid()**.)

Системный вызов `setpgid()`

Прототип системного вызова

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

```
#include <unistd.h>
```

```
int setpgid(pid_t pid, pid_t pgid);
```

Описание системного вызова.

Системный вызов **setpgid** служит для перевода процесса из одной группы процессов в другую, а также для создания новой группы процессов.

Параметр **pid** является идентификатором процесса, который нужно перевести в другую группу, а параметр **pgid** – идентификатором группы процессов, в которую предстоит перевести этот процесс. Не все комбинации этих параметров разрешены. Перевести в другую группу процесс может либо сам себя (и то не во всякую, и не всегда), либо свой процесс-ребенок, который не выполнял системный вызов **exec()**, т. е. не запускал на выполнение другую программу.

- Если параметр **pid** равен **0**, то считается, что процесс переводит в другую группу сам себя.
- Если параметр **pgid** равен **0**, то в Linux считается, что процесс переводится в группу с идентификатором, совпадающим с идентификатором процесса, определяемого первым параметром.
- Если значения, определяемые параметрами **pid** и **pgid**, равны, то создается новая группа с идентификатором, совпадающим с идентификатором переводимого процесса, состоящая первоначально только из этого процесса.

Переход в другую группу без создания новой группы возможен только в пределах одного сеанса. В новую группу не может перейти процесс, являющийся лидером группы, т. е. процесс, идентификатор которого совпадает с идентификатором его группы.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Системный вызов **setpgrp()**

Прототип системного вызова

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

```
#include <unistd.h>
```

```
int setpgrp(void);
```

Описание системного вызова

Системный вызов **setpgrp** служит для перевода текущего процесса во вновь создаваемую группу процессов, идентификатор которой будет совпадать с идентификатором текущего процесса.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки. Процесс, идентификатор которого совпадает с идентификатором его группы, называется лидером группы. Одно из ограничений на применение вызовов **setpgid()** и **setpgrp()** состоит в том, что лидер группы не может перебраться в другую группу. Каждый сеанс в системе также имеет собственный номер. Для того чтобы узнать его, можно воспользоваться системным вызовом **getsid()**. В разных версиях UNIX на него накладываются различные ограничения. В Linux такие ограничения отсутствуют.

Системный вызов **getsid()**

Прототип системного вызова

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

```
#include <unistd.h>
```

```
pid_t getsid(pid_t pid);
```

Описание системного вызова.

Системный вызов возвращает идентификатор сеанса для процесса с идентификатором `pid`. Если параметр `pid` равен 0, то возвращается идентификатор сеанса для данного процесса.

Использование системного вызова **setsid()** приводит к созданию новой группы, состоящей только из процесса, который его выполнил (он становится лидером новой группы), и нового сеанса,

идентификатор которого совпадает с идентификатором процесса, сделавшего вызов. Такой процесс называется лидером сеанса. Этот системный вызов может применять только процесс, не являющийся лидером группы.

Системный вызов `setsid()`

Прототип системного вызова

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

```
#include <unistd.h>
```

```
int setsid(void);
```

Описание системных вызовов

Этот системный вызов может применять только процесс, не являющийся лидером группы, т.е. процесс, идентификатор которого не совпадает с идентификатором его группы. Использование системного вызова `setsid` приводит к созданию новой группы, состоящей только из процесса, который его выполнил (он становится лидером новой группы), и нового сеанса, идентификатор которого совпадает с идентификатором процесса, сделавшего вызов.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Если сеанс имеет управляющий терминал, то этот терминал обязательно приписывается к некоторой группе процессов, входящей в сеанс. Такая группа процессов называется текущей группой процессов для данного сеанса. Все процессы, входящие в текущую группу процессов, могут совершать операции ввода-вывода, используя управляющий терминал. Все остальные группы процессов сеанса называются фоновыми группами, а процессы, входящие в них – фоновыми процессами. При попытке ввода-вывода фонового процесса через управляющий терминал этот процесс получит сигналы, которые стандартно приводят к прекращению работы процесса. Передавать управляющий терминал от одной группы процессов к другой может только лидер сеанса. Для сеансов, не имеющих управляющего терминала, все процессы являются фоновыми.

При завершении работы процесса-лидера сеанса все процессы из текущей группы сеанса получают сигнал **SIGHUP**, который при стандартной обработке приведет к их завершению. Таким образом, после завершения лидера сеанса в нормальной ситуации работу продолжат только фоновые процессы. Процессы, входящие в текущую группу сеанса, могут получать сигналы, инициируемые нажатием определенных клавиш на терминале – **SIGINT** при нажатии клавиш **<CTRL>** и **<c>**, и **SIGQUIT** при нажатии клавиш **<CTRL>** и **<4>**. Стандартная реакция на эти сигналы – завершение процесса (с образованием core файла для сигнала **SIGQUIT**). Необходимо ввести еще одно понятие, связанное с процессом, – эффективный идентификатор пользователя. Каждый пользователь в системе имеет собственный идентификатор – **UID**. Каждый процесс, запущенный пользователем, задействует этот **UID** для определения своих полномочий. Однако иногда, если у исполняемого файла были выставлены соответствующие атрибуты, процесс может выдать себя за процесс, запущенный другим пользователем. Идентификатор пользователя, от имени которого процесс пользуется полномочиями, и является эффективным идентификатором пользователя для процесса – **EUID**. За исключением выше оговоренного случая эффективный идентификатор пользователя совпадает с идентификатором пользователя, создавшего процесс.

9.7 Системный вызов **kill()** и команда **kill**

Из всех перечисленных ранее в разделе «9.4 Аппаратные прерывания (**interrupt**), исключения (**exception**), программные прерывания (**trap**, **software interrupt**). Их обработка» источников сигнала пользователю доступны только два – команда **kill** и посылка сигнала процессу с помощью системного вызова **kill()**.

Команда **kill** обычно используется в следующей форме:

kill [-номер] pid

Здесь **pid** – это идентификатор процесса, которому посылается сигнал, а номер – номер сигнала, который посылается процессу. Послать сигнал (если у вас нет полномочий суперпользователя) можно только процессу, у которого эффективный идентификатор пользователя совпадает с идентификатором пользователя, посылающего сигнал. Если параметр-номер отсутствует, то посылается сигнал **SIGTERM** ,

обычно имеющий номер 15, и реакция на него по умолчанию – завершить работу процесса, который получил сигнал.

Команда kill

Синтаксис команды

kill [-signal] [--] pid

kill -l

Описание команды

Команда kill предназначена для передачи сигнала одному или нескольким специфицированным процессам в рамках полномочий пользователя. Параметр pid определяет процесс или процессы, которым будут доставляться сигналы. Он может быть задан одним из следующих четырех способов:

- Число $n > 0$ – определяет идентификатор процесса, которому будет доставлен сигнал.
- Число 0 – сигнал будет доставлен всем процессам текущей группы для данного управляющего терминала.
- Число -1 с предваряющей опцией -- – сигнал будет доставлен (если позволяют полномочия) всем процессам с идентификаторами, большими 1.
- Число $n < 0$, где n не равно -1, с предваряющей опцией -- – сигнал будет доставлен всем процессам из группы процессов, идентификатор которой равен $-n$.

Параметр **-signal** определяет тип сигнала, который должен быть доставлен, и может задаваться в числовой или символьной форме, например **-9** или **-SIGKILL**. Если этот параметр опущен, процессам по умолчанию посылается сигнал SIGTERM. Послать сигнал (не имея полномочий суперпользователя) можно только процессу, у которого эффективный идентификатор пользователя совпадает с идентификатором пользователя, посылающего сигнал.

Опция **-l** используется для получения списка сигналов, существующих в системе в символьной и числовой формах.

Во многих операционных системах предусмотрены еще и дополнительные опции для команды kill. При использовании системного вызова kill() послать сигнал (не имея полномочий суперпользователя) можно только процессу или процессам, у которых

эффективный идентификатор пользователя совпадает с эффективным идентификатором пользователя процесса, посылающего сигнал.

Системный вызов `kill()`

Прототип системного вызова

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

```
#include <signal.h>
```

```
int kill(pid_t pid, int signal);
```

Описание системного вызова

Системный вызов **kill()** предназначен для передачи сигнала одному или нескольким специфицированным процессам в рамках полномочий пользователя. Послать сигнал (не имея полномочий суперпользователя) можно только процессу, у которого эффективный идентификатор пользователя совпадает с эффективным идентификатором пользователя для процесса, посылающего сигнал.

Аргумент **pid** описывает, кому посылается сигнал, а аргумент **sig** – какой сигнал посылается. Этот системный вызов умеет делать много разных вещей в зависимости от значения аргументов:

- Если `pid > 0` и `sig > 0`, то сигнал номером `sig` (если позволяют привилегии) посылается процессу с идентификатором `pid`.
- Если `pid = 0`, а `sig > 0`, то сигнал с номером `sig` посылается всем процессам в группе, к которой принадлежит посылающий процесс.
- Если `pid = -1`, `sig > 0` и посылающий процесс не является процессом суперпользователя, то сигнал посылается всем процессам в системе, для которых идентификатор пользователя совпадает с эффективным идентификатором пользователя процесса, посылающего сигнал.
- Если `pid = -1`, `sig > 0` и посылающий процесс является процессом суперпользователя, то сигнал посылается всем процессам в системе, за исключением системных процессов (обычно всем, кроме процессов с `pid = 0` и `pid = 1`).
- Если `pid < 0`, но не `-1`, `sig > 0`, то сигнал посылается всем процессам из группы, идентификатор которой равен абсолютному значению аргумента `pid` (если позволяют привилегии).

- Если значение `sig = 0`, то производится проверка на ошибку, а сигнал не посылается, так как все сигналы имеют номера > 0 . Это можно использовать для проверки правильности аргумента `pid` (есть ли в системе процесс или группа процессов с соответствующим идентификатором).

Возвращаемое значение

Системный вызов возвращает 0 при нормальном завершении и -1 при ошибке.

Программа 9-01. Процесс порождает ребенка, и они оба зацикливаются.

Тривиальная программа для иллюстрации понятий «группа процессов», «сеанс», «фоновая группа» и т.д.

```
#include <unistd.h>
int main(void)
{
    (void)fork();
    while(1);
    return 0;
}
```

Листинг 9-01. Программа иллюстрации понятий «группа процессов», «сеанс», «фоновая группа» и т. д.

Используем команду **ps** с опциями **-e** и **j**, которая позволяет получить информацию обо всех процессах в системе и узнать их идентификаторы, идентификаторы групп процессов и сеансов, управляющий терминал сеанса и к какой группе процессов он приписан. Набрав команду "**ps -e j**" (обратите внимание на наличие пробела между буквами **e** и **j**!!!) получим список всех процессов в системе. Колонка **PID** содержит идентификаторы процессов, колонка **PGID** – идентификаторы групп, к которым они принадлежат, колонка **SID** – идентификаторы сеансов, колонка **TTY** – номер соответствующего управляющего терминала, колонка **TPGID** (может присутствовать не во всех версиях UNIX, но в Linux есть) – к какой группе процессов приписан управляющий терминал.

9.8 Системный вызов signal(). Установка собственного обработчика сигнала

Одним из способов изменения поведения процесса при получении сигнала в операционной системе UNIX является использование системного вызова signal().

Системный вызов signal()

Прототип системного вызова

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

Описание системного вызова

Системный вызов signal служит для изменения реакции процесса на какой-либо сигнал. Хотя прототип системного вызова выглядит довольно пугающе, ничего страшного в нем нет. Приведенное выше описание можно словесно изложить следующим образом: функция signal, возвращающая указатель на функцию с одним параметром типа int, которая ничего не возвращает, и имеющая два параметра: параметр sig типа int и параметр handler, служащий указателем на ничего не возвращающую функцию с одним параметром типа int.

Параметр sig – это номер сигнала, обработку которого предстоит изменить.

Параметр handler описывает новый способ обработки сигнала – это может быть указатель на пользовательскую функцию – обработчик сигнала, специальное значение SIG_DFL или специальное значение SIG_IGN. Специальное значение SIG_IGN используется для того, чтобы процесс игнорировал поступившие сигналы с номером sig, специальное значение SIG_DFL – для восстановления реакции процесса на этот сигнал по умолчанию.

Возвращаемое значение

Системный вызов возвращает указатель на старый способ обработки сигнала, значение которого можно использовать для восстановления старого способа в случае необходимости.

Этот системный вызов имеет два параметра: один из них задает номер сигнала, реакцию процесса на который требуется изменить, а второй определяет, как именно мы собираемся ее менять.

Для первого варианта реакции процесса на сигнал – его игнорирования – применяется специальное значение этого параметра **SIG_IGN**.

Например, если требуется игнорировать сигнал **SIGINT**, начиная с некоторого места работы программы, в этом месте программы мы должны употребить конструкцию

```
(void) signal(SIGINT, SIG_IGN);
```

Для второго варианта реакции процесса на сигнал – восстановления его обработки по умолчанию – применяется специальное значение этого параметра **SIG_DFL**.

Для третьего варианта реакции процесса на сигнал в качестве значения параметра подставляется указатель на пользовательскую функцию обработки сигнала, которая должна иметь прототип вида

```
void *handler(int);
```

Ниже приведен пример скелета конструкции для пользовательской обработки сигнала **SIGHUP**.

```
void *my_handler(int nsig)  
{  
    <обработка сигнала>  
}  
int main()  
{  
    ...  
    (void)signal(SIGHUP, my_handler);  
    ...  
}
```

В качестве значения параметра в пользовательскую функцию обработки сигнала (в скелете – параметр **nsig**) передается номер возникшего сигнала, так что одна и та же функция может быть использована для обработки нескольких сигналов.

Приведем пример программы, игнорирующей сигнал SIGINT

Программа 9-02. Программа, игнорирующая сигнал SIGINT

```
#include <signal.h>
```

```
int main(void)
```

```
{
```

```
    /* Выставление реакции процесса на сигнал SIGINT на  
    игнорирование */
```

```
    (void)signal(SIGINT, SIG_IGN);
```

```
    /*Начиная с этого места, процесс будет игнорировать  
    возникновение сигнала SIGINT */
```

```
    while(1);
```

```
    return 0;
```

```
}
```

Листинг 9-02. Программа, игнорирующая сигнал SIGINT

Эта программа не делает ничего полезного, кроме переустановки реакции на нажатие клавиш <CTRL> и <C> на игнорирование возникающего сигнала и своего бесконечного зацикливания.

Другой пример программы с пользовательской обработкой сигнала SIGINT

Программа 9-03. Программа с пользовательской обработкой сигнала SIGINT

```
#include <signal.h>
```

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

```
#include <iostream>
```

```
using namespace std;
```

```
/* Функция my_handler – пользовательский обработчик сигнала */
```

```
void my_handler(int nsig)
```

```
{
```

```
    cout<< "Receive signal " << nsig << "CTRL-C pressed" << endl;
```

```
}
```

```
int main(void)
```

```
{
```

```
    /* Выставление реакции процесса на сигнал SIGINT */
```

```
    (void)signal(SIGINT, my_handler);
```

```

    /*Начиная с этого места, процесс будет печатать сообщение о
    возникновении сигнала SIGINT */
    while(1);
    return 0;
}

```

Листинг 9-03. Программа с пользовательской обработкой сигнала SIGINT

Эта программа отличается от программы из раздела «Прогон программы, игнорирующей сигнал SIGINT» тем, что в ней введена обработка сигнала SIGINT пользовательской функцией.

До сих пор в примерах игнорировали значение, возвращаемое системным вызовом signal(). На самом деле этот системный вызов возвращает указатель на предыдущий обработчик сигнала, что позволяет восстанавливать переопределенную реакцию на сигнал.

Программа 9-04. Программа с пользовательской обработкой сигнала SIGINT, возвращающаяся к первоначальной реакции на этот сигнал после пяти его обработок

```

#include <signal.h>
#include <stdio.h>
#include <iostream>
using namespace std;
int i=0; /* Счетчик числа обработок сигнала */
void (*p)(int); /* Указатель, в который будет занесен адрес
предыдущего обработчика сигнала */
/* Функция my_handler – пользовательский обработчик сигнала */
void my_handler(int nsig)
{
    cout<< "Receive signal " << nsig << "CTRL-C pressed" << endl;
    i = i+1;
    /* После 5-й обработки возвращение первоначальной реакции на
    сигнал */
    if(i == 5) (void)signal(SIGINT, p);
}
int main(void)

```

```

{
    /* Выставление своей реакции процесса на сигнал SIGINT,
запоминая адрес предыдущего обработчика */
    p = signal(SIGINT, my_handler);
    /*Начиная с этого места, процесс будет 5 раз печатать сообщение
о возникновении сигнала SIGINT */
    while(1);
    return 0;
}

```

Листинг 9-04. Программа с пользовательской обработкой сигнала SIGINT.

Сигналы SIGUSR1 и SIGUSR2. Использование сигналов для синхронизации процессов

В операционной системе UNIX существует два сигнала, источниками которых могут служить только системный вызов kill() или команда kill, – это сигналы SIGUSR1 и SIGUSR2. Обычно их применяют для передачи информации о произошедшем событии от одного пользовательского процесса другому в качестве сигнального средства связи. При реализации нитей исполнения в операционной системе Linux сигналы SIGUSR1 и SIGUSR2 используются для организации синхронизации между процессами, представляющими нити исполнения, и процессом-координатором в служебных целях. Поэтому пользовательские программы, применяющие в своей работе нити исполнения, не могут задействовать сигналы SIGUSR1 и SIGUSR2 .

9.10 Завершение порожденного процесса. Системный вызов waitpid(). Сигнал SIGCHLD

В материалах семинара 3 (раздел «Завершение процесса. Функция exit()») при изучении завершения процесса говорилось о том, что если процесс-ребенок завершает свою работу прежде процесса-родителя и процесс-родитель явно не указал, что он не заинтересован в получении информации о статусе завершения процесса-ребенка, то завершившийся процесс не исчезает из системы окончательно, а остается в состоянии «закончил исполнение» (зомби-процесс) либо до

завершения процесса-родителя, либо до того момента, когда родитель сообразовит получить эту информацию. Для получения такой информации процесс-родитель может воспользоваться системным вызовом `waitpid()` или его упрощенной формой `wait()`. Системный вызов `waitpid()` позволяет процессу-родителю синхронно получить данные о статусе завершившегося процесса-ребенка либо блокируя процесс-родитель до завершения процесса-ребенка, либо без блокировки при его периодическом вызове с опцией `WNOHANG`. Эти данные занимают 16 бит и в рамках нашего курса могут быть расшифрованы следующим образом:

Если процесс завершился при помощи явного или неявного вызова функции `exit()`, то данные выглядят так (старший бит находится слева):



Рис. 9.2. Результат вызова `waitpid()` при завершении процесса с помощью явного или неявного вызова функции `exit()`

Если процесс был завершён сигналом, то данные выглядят так (старший бит находится слева):



Рис. 9.3. Результат вызова `waitpid()` при завершении процесса с помощью сигнала

Каждый процесс-ребенок при завершении работы посылает своему процессу-родителю специальный сигнал `SIGCHLD`, на который у всех процессов по умолчанию установлена реакция "игнорировать сигнал". Наличие такого сигнала совместно с системным вызовом `waitpid()` позволяет организовать асинхронный сбор информации о статусе завершившихся порожденных процессов процессом-родителем.

Системные вызовы wait() и waitpid()

Прототипы системных вызовов

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

```
#include <wait.h>
```

```
pid_t waitpid(pid_t pid, int *status,  
              int options);
```

```
pid_t wait(int *status);
```

Описание системных вызовов

Это описание не является полным описанием системных вызовов. Для получения полного описания необходимо обратиться к UNIX Manual. Системный вызов waitpid() блокирует выполнение текущего процесса до тех пор, пока либо не завершится порожденный им процесс, определяемый значением параметра pid, либо текущий процесс не получит сигнал, для которого установлена реакция по умолчанию «завершить процесс» или реакция обработки пользовательской функцией. Если порожденный процесс, заданный параметром pid, к моменту системного вызова находится в состоянии «закончил исполнение», то системный вызов возвращается немедленно без блокирования текущего процесса.

Параметр pid определяет порожденный процесс, завершения которого дожидается процесс-родитель, следующим образом:

- Если `pid > 0`, ожидаем завершения процесса с идентификатором pid.
- Если `pid = 0`, ожидаем завершения любого порожденного процесса в группе, к которой принадлежит процесс-родитель.
- Если `pid = -1`, ожидаем завершения любого порожденного процесса.
- Если `pid < 0`, но не `-1`, ожидаем завершения любого порожденного процесса из группы, идентификатор которой равен абсолютному значению параметра pid.
-

Параметр **options** может принимать два значения: **0** и **WNOHANG**. Значение **WNOHANG** требует немедленного возврата из вызова без блокировки текущего процесса в любом случае. Если системный вызов обнаружил завершившийся порожденный процесс из

числа специфицированных параметром **pid**, то этот процесс удаляется из вычислительной системы, а по адресу, указанному в параметре **status**, сохраняется информация о статусе его завершения. Параметр **status** может быть задан равным **NULL**, если эта информация не имеет для нас значения. При обнаружении завершившегося процесса системный вызов возвращает его идентификатор. Если вызов был сделан с установленной опцией **WNOHANG** и порожденный процесс, специфицированный параметром **pid**, существует, но еще не завершился, системный вызов вернет значение **0**. Во всех остальных случаях он возвращает отрицательное значение. Возврат из вызова, связанный с возникновением обработанного пользователем сигнала, может быть в этом случае идентифицирован по значению системной переменной **errno == EINTR**, и вызов может быть сделан снова.

Системный вызов **wait** является синонимом для системного вызова **waitpid** со значениями параметров **pid = -1**, **options = 0**.

Используя системный вызов **signal()**, мы можем явно установить игнорирование этого сигнала (**SIG_IGN**), тем самым проинформировав систему, что нас не интересует, каким образом завершатся порожденные процессы. В этом случае зомби-процессов возникать не будет, но и применение системных вызовов **wait()** и **waitpid()** будет запрещено.

Программа 9-05 с асинхронным получением информации о статусе завершения порожденного процесса.

Программа с асинхронным получением информации о статусе двух завершившихся порожденных процессов

```
#include <sys/types.h>
#include <unistd.h>
#include <wait.h>
#include <signal.h>
#include <stdio.h>
#include <iostream>
using namespace std;
/* Функция my_handler – обработчик сигнала SIGCHLD */
void my_handler(int nsig)
```

```

{
    int status;
    pid_t pid;
    /* Опрашиваем статус завершившегося процесса и одновременно
    узнаем его идентификатор */
    if((pid = waitpid(-1, &status, 0)) < 0)
    {
        /* Если возникла ошибка – сообщение о ней и продолжение
        работы */
        cout<<"Some error on waitpid errno "<< errno<< endl;
    }
    else
    {
        /* Иначе анализирование статуса завершившегося процесса */
        if ((status & 0xff) == 0)
        {
            /* Процесс завершился с явным или неявным вызовом функции
            exit() */
            printf("Process %d was exited with status %d\n",pid, status >> 8);
        }
        else if ((status & 0xff00) == 0)
        {
            /* Процесс был завершён с помощью сигнала */
            printf("Process %d killed by signal %d %s\n",  pid, status
            & 0x7f,(status & 0x80) ? "with core file" : "without core file");
        }
    }
}

int main(void)
{
    pid_t pid;
    /* Установление обработчика для сигнала SIGCHLD */
    (void) signal(SIGCHLD, my_handler);
    /* Порождение Child 1 */
    if((pid = fork()) < 0)
    {
        cout<<"Can't fork child 1"<< endl;
        exit(1);
    }
}

```



```

    }
    else if (pid == 0)
    {
        /* Child 1 – завершается с кодом 200 */
        exit(200);
    }
    /* Продолжение процесса-родителя – порождаем Child 2 */
    if((pid = fork()) < 0)
    {
        cout<<"Can't fork child 2"<< endl;
        exit(1);
    }
    else if (pid == 0)
    {
        /* Child 2 – циклится, необходимо удалять с помощью сигнала! */
        while(1);
    }
    /* Продолжение процесса-родителя – уходим в цикл */
    while(1);
    return 0;
}

```

Листинг 9-05. Программа 8-05 с асинхронным получением информации о статусе двух завершившихся порожденных процессов

В этой программе родитель порождает два процесса. Один из них завершается с кодом 200, а второй закичивается. Перед порождением процессов родитель устанавливает обработчик прерывания для сигнала SIGCHLD, а после их порождения уходит в бесконечный цикл. В обработчике прерывания вызывается waitpid() для любого порожденного процесса. Так как в обработчик попадаем, когда какой-либо из процессов завершился, системный вызов не блокируется, и можно получить информацию об идентификаторе завершившегося процесса и причине его завершения.

9.11 Возникновение сигнала SIGPIPE при попытке записи в pipe или FIFO, который никто не собирается читать

Для pipe и FIFO системные вызовы read() и write() имеют определенные особенности поведения. Одной из таких особенностей является получение сигнала SIGPIPE процессом, который пытается записывать информацию в pipe или в FIFO в том случае, когда читать ее уже некому (нет ни одного процесса, который держит соответствующий pipe или FIFO открытым для чтения). Реакция по умолчанию на этот сигнал – прекратить работу процесса. В процессе изучения курса мы видели ряд системных вызовов, которые могут во время выполнения блокировать процесс. К их числу относятся системный вызов open() при открытии FIFO, системные вызовы read() и write() при работе с pipe'ами и FIFO, системные вызовы msgsnd() и msgrcv() при работе с очередями сообщений, системный вызов semop() при работе с семафорами и т. д. Что произойдет с процессом, если он, выполняя один из этих системных вызовов, получит какой-либо сигнал? Дальнейшее поведение процесса зависит от установленной для него реакции на этот сигнал.

- Если реакция на полученный сигнал была «игнорировать сигнал» (независимо от того, установлена она по умолчанию или пользователем с помощью системного вызова signal()), то поведение процесса не изменится.
- Если реакция на полученный сигнал установлена по умолчанию и заключается в прекращении работы процесса, то процесс перейдет в состояние «закончил исполнение».
- Если реакция процесса на сигнал заключается в выполнении пользовательской функции, то процесс выполнит эту функцию (если он находился в состоянии «ожидание», он попадет в состояние «готовность» и затем в состояние «исполнение») и вернется из системного вызова с констатацией ошибочной ситуации (некоторые системные вызовы позволяют операционной системе после выполнения обработки сигнала вновь вернуть процесс в состояние ожидания).

Отличить такой возврат от действительно ошибочной ситуации можно с помощью значения системной переменной errno, которая в

этом случае примет значение **EINTR** (для вызова **write** и сигнала **SIGPIPE** соответствующее значение в порядке исключения будет **EPIPE**).

Чтобы пришедший сигнал **SIGPIPE** не завершил работу процесса по умолчанию, необходимо его обработать самостоятельно (функция-обработчик при этом может быть и пустой!). Но этого мало. Поскольку нормальный ход выполнения системного вызова был нарушен сигналом, возврат из него будет с отрицательным значением, которое свидетельствует об ошибке. Проанализировав значение системной переменной **errno** на предмет совпадения со значением **EPIPE**, можно отличить возникновение сигнала **SIGPIPE** от других ошибочных ситуаций (неправильные значения параметров и т. д.) и продолжить работу программы.

9.12 Понятие о надежности сигналов. POSIX функции для работы с сигналами

Основным недостатком системного вызова **signal()** является его низкая надежность. Во многих вариантах операционной системы **UNIX** установленная при его помощи обработка сигнала пользовательской функцией выполняется только один раз, после чего автоматически восстанавливается реакция на сигнал по умолчанию. Для постоянной пользовательской обработки сигнала необходимо каждый раз заново устанавливать реакцию на сигнал прямо внутри функции-обработчика. В системных вызовах и пользовательских программах могут существовать критические участки, на которых процессу недопустимо отвлекаться на обработку сигналов. Можно выставить на этих участках реакцию «игнорировать сигнал» с последующим восстановлением предыдущей реакции, но если сигнал все-таки возникнет на критическом участке, то информация о его возникновении будет безвозвратно потеряна. Наконец, последний недостаток связан с невозможностью определения количества сигналов одного и того же типа, поступивших процессу, пока он находился в состоянии готовности. Сигналы одного типа в очередь не ставятся! Процесс может узнать о том, что сигнал или сигналы определенного типа были ему переданы, но не может определить их количество.

Этот недостаток можно проиллюстрировать, слегка изменив программу с асинхронным получением информации о статусе завершившихся процессов, рассмотренную ранее в разделе «Изучение особенностей получения терминальных сигналов текущей и фоновой группой процессов».

В новой программе процесс-родитель порождает в цикле пять новых процессов, каждый из которых сразу же завершается со своим собственным кодом, после чего уходит в бесконечный цикл.

Программа 9-06 для иллюстрации ненадежности сигналов

```
#include <sys/types.h>
#include <unistd.h>
#include <wait.h>
#include <signal.h>
#include <stdio.h>
#include <errno.h>
#include <cstdlib.h>
#include <iostream>
using namespace std;
/* Функция my_handler – обработчик сигнала SIGCHLD */
void my_handler(int nsig)
{
    int status;
    pid_t pid;
    /* Опрашиваем статус завершившегося процесса и одновременно
    узнаем его идентификатор */
    if((pid = waitpid(-1, &status, 0)) < 0)
    {
        /* Если возникла ошибка – сообщение о ней и продолжение
        работы */
        cout<<"Some error on waitpid errno "<< errno << endl;
    }
    else
    {
        /* Иначе анализируем статус завершившегося процесса */
        if ((status & 0xff) == 0)
        {
```

```

        /* Процесс завершился с явным или неявным вызовом функции
exit() */
        printf("Process %d was exited with status %d\n", pid, status >> 8);
    }
    else if ((status & 0xff00) == 0)
    {
        /* Процесс был завершён с помощью сигнала */
        printf("Process %d killed by signal %d %s\n", pid, status
&0x7f,(status & 0x80) ? "with core file" : "without core file");
    }
}
}
int main(void)
{
    pid_t pid;
    int i;
    /* Установление обработчика для сигнала SIGCHLD */
    (void) signal(SIGCHLD, my_handler);
    /* В цикле порождение 5 процессов-детей */
    for (i=0; i < 5; i++)
        if((pid = fork()) < 0)
        {
            cout<< "Can't fork child " << i<< endl;
            exit(1);
        }
    else if (pid == 0)
    {
        /* Child i – завершается с кодом 200 + i */
        exit(200 + i);
    }
    /* Продолжение процесса-родителя – уход на новую итерацию */
}
/* Продолжение процесса-родителя – уход в цикл */
while(1);
return 0;
}

```

Листинг 9-06. Программа для иллюстрации ненадежности сигналов

Последующие версии **System V** и **BSD** пытались устранить эти недостатки собственными средствами. Единый способ более надежной обработки сигналов появился с введением **POSIX** стандарта на системные вызовы **UNIX**. Набор функций и системных вызовов для работы с сигналами был существенно расширен и построен таким образом, что позволял временно блокировать обработку определенных сигналов, не допуская их потери. Однако проблема, связанная с определением количества пришедших сигналов одного типа, по-прежнему остается актуальной. (Необходимо отметить, что подобная проблема существует на аппаратном уровне и для внешних прерываний. Процессор зачастую не может определить, какое количество внешних прерываний с одним номером возникло, пока он выполнял очередную команду.)

Порядок выполнения лабораторной работы:

1. Наберите программу 9-01, откомпилируйте ее и запустите на исполнение (лучше всего из-под оболочки Midnight Commander – mc). Запустив команду "ps -e j" с другого экрана, проанализируйте значения идентификаторов группы процессов, сеансов, прикрепления управляющего терминала, текущей и фоновой групп. Убедитесь, что тривиальные процессы относятся к текущей группе сеанса. Проверьте реакцию текущей группы на сигналы SIGINT – нажатие клавиш <CTRL> и <C> – и SIGQUIT – нажатие клавиш <CTRL> и <4>.
2. Запустите теперь программу 9-01 в фоновом режиме, например командой "a.out". Проанализируйте значения идентификаторов группы процессов, сеансов, прикрепления управляющего терминала, текущей и фоновой групп. Убедитесь, что тривиальные процессы относятся к фоновой группе сеанса. Проверьте реакцию фоновой группы на сигналы SIGINT – нажатие клавиш <CTRL> и <C> – и SIGQUIT – нажатие клавиш <CTRL> и <4>. Ликвидируйте тривиальные процессы с помощью команды kill .
3. Возьмите снова тривиальную программу из предыдущего раздела и запустите ее на исполнение из-под Midnight Commander в текущей группе. Проанализировав значения идентификаторов группы процессов, сеансов, прикрепления

- управляющего терминала, текущей и фоновой групп, ликвидируйте лидера сеанса для тривиальных процессов. Убедитесь, что все процессы в текущей группе этого сеанса прекратили свою работу.
4. Запустите тривиальную программу в фоновом режиме. Снова удалите лидера сеанса для тривиальных процессов. Убедитесь, что фоновая группа продолжает работать. Ликвидируйте тривиальные процессы.
 5. Модифицируйте программу 9-02 так, чтобы она перестала реагировать и на нажатие клавиш <CTRL> и <4>. Откомпилируйте и запустите ее, убедитесь в отсутствии ее реакций на внешние раздражители. Снимать программу придется теперь с другого терминала командой kill.
 6. Модифицируйте программу 9-03 так, чтобы она печатала сообщение и о нажатии клавиш <CTRL> и <4>. Используйте одну и ту же функцию для обработки сигналов SIGINT и SIGQUIT. Откомпилируйте и запустите ее, проверьте корректность работы. Снимать программу также придется с другого терминала командой kill.
 7. Наберите, откомпилируйте программу 9-04 и запустите ее на исполнение
 8. Когда рассматривалась связь родственных процессов через pipe, речь шла о том, что pipe является однонаправленным каналом связи и что для организации связи через один pipe в двух направлениях необходимо задействовать механизмы взаимной синхронизации процессов. Организуйте двустороннюю поочередную связь процесса-родителя и процесса-ребенка через pipe, используя для синхронизации сигналы SIGUSR1 и SIGUSR2, модифицировав программу для организации однонаправленной связи между родственными процессами через pipe семинара 4.
 9. Задача повышенной сложности: организуйте побитовую передачу целого числа между двумя процессами, используя для этого только сигналы SIGUSR1 и SIGUSR2 .
 10. Откомпилируйте программу 9-05 и запустите ее на исполнение. Второй порожденный процесс завершайте с помощью команды kill с каким-либо номером сигнала. Родительский процесс также будет необходимо завершать командой kill.

11. Задача повышенной сложности: модифицируйте обработку сигнала в программе 6, не применяя POSIX-сигналы, так, чтобы процесс-родитель сообщал о статусе всех завершившихся процессов-детей.

Контрольные вопросы

1. Понятие виртуальной файловой систем.
2. Операции над файловыми системами.
3. Монтирование файловых систем. Блочные, символьные устройства.
4. Понятие драйвера. Блочные, символьные драйверы, драйверы низкого уровня.
5. Файловый интерфейс. Аппаратные прерывания (interrupt), исключения (exception), программные прерывания (trap, software interrupt). Их обработка.
6. Понятие сигнала. Способы возникновения сигналов и виды их обработки.
7. Понятия группы процессов, сеанса, лидера группы, лидера сеанса, управляющего терминала сеанса.
8. Системные вызовы `getpgrp()`, `setpgrp()`, `getpgid()`, `setpgid()`, `getsid()`, `setsid()`.
9. Системный вызов `kill()` и команда `kill`.
10. Изучение особенностей получения терминальных сигналов текущей и фоновой группой процессов.
11. Системный вызов `signal()`. Установка собственного обработчика сигнала.
12. Сигналы `SIGUSR1` и `SIGUSR2`. Использование сигналов для синхронизации процессов.
13. Завершение порожденного процесса. Системный вызов `waitpid()`. Сигнал `SIGCHLD`.
14. Возникновение сигнала `SIGPIPE` при попытке записи в `pipe` или `FIFO`.
15. Понятие о надежности сигналов.
16. POSIX функции для работы с сигналами.

Лабораторная работа 10

СЕМЕЙСТВО ПРОТОКОЛОВ TCP/IP.

СОКЕТЫ (sockets) В UNIX И ОСНОВЫ РАБОТЫ С НИМИ

Цели и задачи

Изучить семейство протоколов TCP/IP. Рассмотреть использование модели клиент-сервер для взаимодействия удаленных процессов. Научиться организации связи между удаленными процессами с помощью датаграмм. Рассмотреть функции преобразования IP-адресов `inet_ntoa()`, `inet_aton()`. Научиться использовать основные функции и системные вызовы для работы с сокетами.

10.1 Краткая история семейства протоколов TCP/IP

Все многообразие сетевых приложений и многомиллионная всемирная компьютерная сеть выросли из четырехкомпьютерной сети ARPANET, созданной по заказу Министерства обороны США и связавшей вычислительные комплексы в Стэнфордском исследовательском институте, Калифорнийском университете в Санта-Барбаре, Калифорнийском университете в Лос-Анджелесе и университете Юты. Первая передача информации между двумя компьютерами сети ARPANET состоялась в октябре 1969 года, и эту дату принято считать датой рождения нелокальных компьютерных сетей. (Необходимо отметить, что дата является достаточно условной, так как первая связь двух удаленных компьютеров через коммутируемые телефонные линии была осуществлена еще в 1965 году, а реальные возможности для разработки пользователями ARPANET сетевых приложений появились только в 1972 году.) При создании ARPANET был разработан протокол сетевого взаимодействия коммуникационных узлов – Network Control Protocol (NCP), осуществлявший связь посредством передачи датаграмм. Этот протокол был предназначен для конкретного архитектурного построения сети и базировался на предположении, что сеть является статической и настолько надежной, что компьютерам не требуется умения реагировать на возникающие ошибки. По мере роста ARPANET и необходимости подключения к ней сетей, построенных на других архитектурных принципах (пакетные спутниковые сети, наземные пакетные радиосети), от этого предположения пришлось

отказаться и искать другие подходы к построению сетевых систем. Результатом исследований в этих областях стало появление семейства протоколов TCP/IP, на базе которого обеспечивалась надежная доставка информации по неоднородной сети. Это семейство протоколов до сих пор занимает ведущее место в качестве сетевой технологии, используемой в операционной системе UNIX.

10.2 Общие сведения об архитектуре семейства протоколов TCP/IP

Семейство протоколов TCP/IP построено по «слоеному» принципу. Хотя оно и имеет многоуровневую структуру, его строение отличается от строения эталонной модели OSI, предложенной стандартом ISO. Это и неудивительно, так как основные черты семейства TCP/IP были заложены до появления эталонной модели и во многом послужили толчком для ее разработки. В семействе протоколов TCP/IP можно выделить четыре уровня:

- Уровень сетевого интерфейса.
- Уровень Internet.
- Транспортный уровень.
- Уровень приложений/процессов.

Соотношение уровней семейства TCP/IP и уровней модели OSI/ISO приведено на рисунке 10.1.



Рис. 10.1. Соотношение моделей OSI/ISO и TCP/IP

На каждом уровне семейства TCP/IP присутствует несколько протоколов. Связь между наиболее употребительными протоколами и их принадлежность уровням изображены на рисунке 10.2.

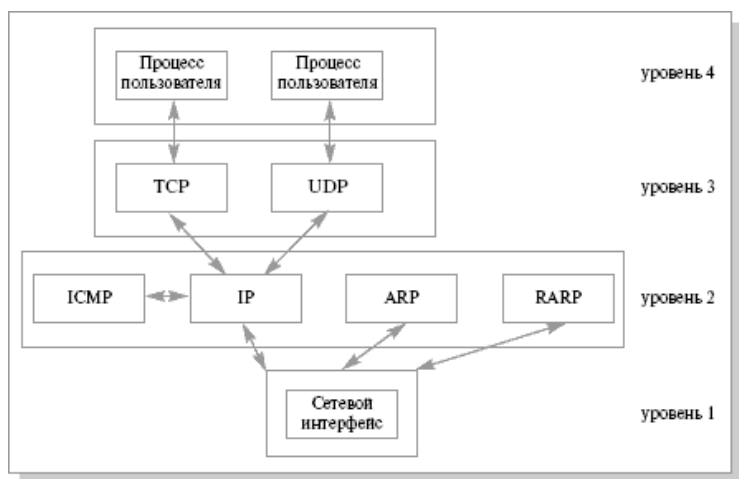


Рис.10.2. Основные протоколы семейства TCP/IP

10.3 Уровень сетевого интерфейса

Уровень сетевого интерфейса составляют протоколы, которые обеспечивают передачу данных между узлами связи, физически напрямую соединенными друг с другом или, иначе говоря, подключенными к одному сегменту сети, и соответствующие физические средства передачи данных. К этому уровню относятся протоколы Ethernet, Token Ring, SLIP, PPP и т.д. и такие физические средства, как витая пара, коаксиальный кабель, оптоволоконный кабель и т.д. Формально протоколы уровня сетевого интерфейса не являются частью семейства TCP/IP, но существующие стандарты определяют, каким образом должна осуществляться передача данных семейства TCP/IP с использованием этих протоколов. На уровне сетевого интерфейса в операционной системе UNIX обычно функционируют драйверы различных сетевых плат.

Передача информации на уровне сетевого интерфейса производится на основании физических адресов, соответствующих точкам входа сети в узлы связи (например, физических адресов

сетевых карт). Каждая точка входа имеет свой уникальный адрес – MAC-адрес (Media Access Control), физически зашитый в нее на этапе изготовления. Так, например, каждая сетевая плата Ethernet имеет собственный уникальный 48-битовый номер.

10.4 Уровень Internet. Протоколы IP, ICMP, ARP, RARP. Internet–адреса

Из многочисленных протоколов уровня Internet перечислены только некоторые:

- **ICMP – Internet Control Message Protocol.** Протокол обработки ошибок и обмена управляющей информацией между узлами сети.
- **IP – Internet Protocol.** Это протокол, который обеспечивает доставку пакетов информации для протокола ICMP и протоколов транспортного уровня TCP и UDP.
- **ARP – Address Resolution Protocol.** Это протокол для отображения адресов уровня Internet в адреса уровня сетевого интерфейса.
- **RARP – Reverse Address Resolution Protocol.** Этот протокол служит для решения обратной задачи: отображения адресов уровня сетевого интерфейса в адреса уровня Internet.

Два последних протокола используются не для всех сетей; только некоторые сети требуют их применения.

Уровень Internet обеспечивает доставку информации от сетевого узла отправителя к сетевому узлу получателя без установления виртуального соединения с помощью датаграмм и не является надежным. Центральным протоколом уровня является протокол IP. Вся информация, поступающая к нему от других протоколов, оформляется в виде IP-пакетов данных (IP datagrams). Каждый IP-пакет содержит адреса компьютера отправителя и компьютера получателя, поэтому он может передаваться по сети независимо от других пакетов и, возможно, по своему собственному маршруту. Любая ассоциативная связь между пакетами, предполагающая знания об их содержании, должна осуществляться на более высоком уровне семейства протоколов.

IP-уровень семейства TCP/IP не является уровнем, обеспечивающим надежную связь, так как он не гарантирует ни

доставку отправленного пакета информации, ни то, что пакет будет доставлен без ошибок. IP вычисляет и проверяет контрольную сумму, которая покрывает только его собственный 20-байтовый заголовок для пакета информации (включающий, например, адреса отправителя и получателя). Если IP-заголовок пакета при передаче оказывается испорченным, то весь пакет просто отбрасывается. Ответственность за повторную передачу пакета тем самым возлагается на вышестоящие уровни. IP-протокол, при необходимости, осуществляет фрагментацию и дефрагментацию данных, передаваемых по сети. Если размер IP-пакета слишком велик для дальнейшей передачи по сети, то полученный пакет разбивается на несколько фрагментов, и каждый фрагмент оформляется в виде нового IP-пакета с теми же адресами отправителя и получателя. Фрагменты собираются в единое целое только в конечной точке своего путешествия. Если при дефрагментации пакета обнаруживается, что хотя бы один из фрагментов был потерян или отброшен, то отбрасывается и весь пакет целиком. Уровень Internet отвечает за маршрутизацию пакетов. Для обмена информацией между узлами сети в случае возникновения проблем с маршрутизацией пакетов используется протокол ICMP. С помощью сообщений этого же протокола уровень Internet умеет частично управлять скоростью передачи данных – он может попросить отправителя уменьшить скорость передачи.

Поскольку на уровне Internet информация передается от компьютера-отправителя к компьютеру-получателю, ему требуются специальные IP-адреса компьютеров (а точнее, их точек подсоединения к сети – сетевых интерфейсов) – удаленные части полных адресов процессов. Далее рассматривается IP версии 4 (IPv4), которая предполагает наличие у каждого сетевого интерфейса уникального 32-битового адреса. Когда разрабатывалось семейство протоколов TCP/IP, казалось, что 32 бита адреса будет достаточно для всех нужд сети, однако не прошло и 30 лет, как выяснилось, что этого мало. Поэтому была разработана версия 6 для IP (IPv6), предполагающая наличие 128-битовых адресов. С точки зрения сетевого программиста IPv6 мало отличается от IPv4, но имеет более сложный интерфейс передачи параметров. Все IP-адреса версии 4 принято делить на пять классов. Принадлежность адреса к некоторому классу определяют по количеству последовательных 1 в старших битах адреса (рис. 10.3). Адреса классов А, В и С используют

собственно для адресации сетевых интерфейсов. Адреса класса D применяются для групповой рассылки информации (multicast addresses) и далее нас интересовать не будут. Класс E (про который во многих книгах по сетям забывают) был зарезервирован для будущих расширений. Каждый из IP-адресов классов А–С логически делится на две части: идентификатор или номер сети и идентификатор или номер узла в этой сети. Идентификаторы сетей в настоящее время присваиваются локальным сетям специальной международной организацией – корпорацией Internet по присвоению имен и номеров (ICANN). Присвоение адреса конкретному узлу сети, получившей идентификатор, является заботой ее администратора. Класс А предназначен для небольшого количества сетей, содержащих очень много компьютеров, класс С – напротив, для большого количества сетей с малым числом компьютеров. Класс В занимает среднее положение. Надо отметить, что все идентификаторы сетей классов А и В к настоящему моменту уже задействованы.

Любая организация, которой был выделен идентификатор сети из любого класса, может произвольным образом разделить имеющееся у нее адресное пространство идентификаторов узлов для создания подсетей. Если предположить, что выделен адрес сети класса С, в котором под номер узла сети отведено 8 бит.

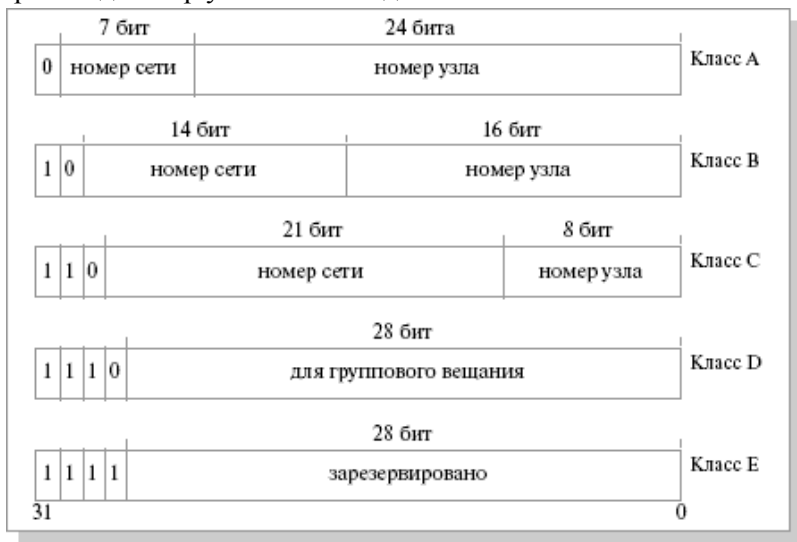


Рис.10.3. Классы IP-адресов

Если нужно присвоить IP-адреса 100 компьютерам, которые организованы в 10 Ethernet-сегментов по 10 компьютеров в каждом, можно поступить по-разному. Можно присвоить номера от 1 до 100 компьютерам, игнорируя их принадлежность к конкретному сегменту – воспользовавшись стандартной формой IP-адреса. Или же можно выделить несколько старших бит из адресного пространства идентификаторов узлов для идентификации сегмента сети, например 4 бита, а для адресации узлов внутри сегмента использовать оставшиеся 4 бита. Последний способ получил название адресации с использованием подсетей (рис. 10.4).

Запоминать четырехбайтовые числа для человека достаточно сложно, поэтому принято записывать IP-адреса в символической форме, переводя значение каждого байта в десятичный вид по отдельности и разделяя полученные десятичные числа в записи точками, начиная со старшего байта: 192.168.253.10.

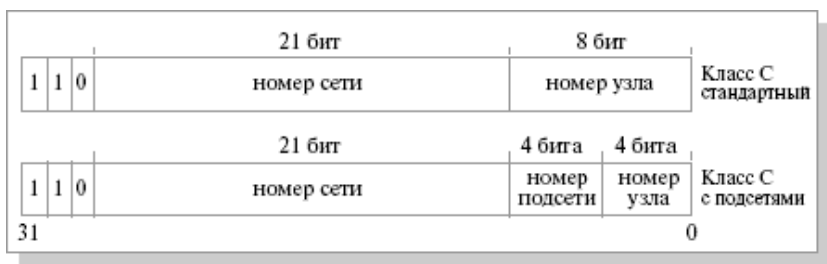


Рис. 10.4. Адресация с подсетями

Допустим, что мы имеем дело с сегментом сети, использующим Ethernet на уровне сетевого интерфейса и состоящим из компьютеров, где применяются протоколы TCP/IP на более высоких уровнях. Тогда в сети есть два вида адресов: 48-битовые физические адреса Ethernet (MAC-адреса) и 32-битовые IP-адреса. Для нормальной передачи информации необходимо, чтобы Internet уровень семейства протоколов, обращаясь к уровню сетевого интерфейса, знал, какой физический адрес соответствует данному IP-адресу и наоборот, т. е. умел «разрешать адреса». При разрешении адресов может возникнуть две сложности:

- Если IP-адреса компьютеров, которым или через необходимо передать данные, известны, то каким образом Internet уровень семейства протоколов TCP/IP сможет определить

соответствующие им MAC-адреса? Эта проблема получила название address resolution problem (проблема разрешения адресов).

- Пусть есть бездисковые рабочие станции или рабочие станции, на которых операционные системы сгенерированы без назначения IP-адресов (это часто делается, когда один и тот же образ операционной системы загружается на ряд компьютеров, например, в учебных классах). Тогда при старте операционной системы на каждом таком компьютере операционная система знает только MAC-адреса, соответствующие данному компьютеру. Как можно определить, какой Internet-адрес был выделен данной рабочей станции? Эта проблема называется reverse address resolution problem (обратная проблема разрешения адресов).

Первая задача решается с использованием протокола ARP, вторая – с помощью протокола RARP.

Протокол ARP позволяет компьютеру разослать специальное сообщение по всему сегменту сети, которое требует от компьютера, имеющего содержащийся в сообщении IP-адрес, откликнуться и указать свой физический адрес. Это сообщение поступает всем компьютерам в сегменте сети, но откликается на него только тот, кого спрашивали. После получения ответа запрашивавший компьютер может установить необходимое соответствие между IP-адресом и MAC-адресом.

Для решения второй проблемы один или несколько компьютеров в сегменте сети должны выполнять функции RARP-сервера и содержать набор физических адресов для рабочих станций и соответствующих им IP-адресов. Когда рабочая станция с операционной системой, сгенерированной без назначения IP-адреса, начинает свою работу, она получает MAC-адрес от сетевого оборудования и рассылает соответствующий RARP-запрос, содержащий этот адрес, всем компьютерам сегмента сети. Только RARP-сервер, содержащий информацию о соответствии указанного физического адреса и выделенного IP-адреса, откликается на данный запрос и отправляет ответ, содержащий IP-адрес.

10.5 Транспортный уровень. Протоколы TCP и UDP. TCP и UDP сокет. Адресные пространства портов. Понятие encapsulation

К протоколам транспортного уровня относятся протоколы TCP и UDP. Протокол TCP реализует потоковую модель передачи информации, хотя в его основе, как и в основе протокола UDP, лежит обмен информацией через пакеты данных. Он представляет собой ориентированный на установление логической связи (connection-oriented), надежный (обеспечивающий проверку контрольных сумм, передачу подтверждения в случае правильного приема сообщения, повторную передачу пакета данных в случае неполучения подтверждения в течение определенного промежутка времени, правильную последовательность получения информации, полный контроль скорости передачи данных) дуплексный способ связи между процессами в сети. Протокол UDP, наоборот, является способом связи ненадежным, ориентированным на передачу сообщений (датаграмм). От протокола IP он отличается двумя основными чертами: использованием для проверки правильности принятого сообщения контрольной суммы, насчитанной по всему сообщению, и передачей информации не от узла сети к другому узлу, а от отправителя к получателю. Полный адрес удаленного процесса или промежуточного объекта для конкретного способа связи с точки зрения операционных систем определяется парой адресов: <числовой адрес компьютера в сети, локальный адрес>.

Такая пара получила название socket (гнездо, панель), так как по сути дела является виртуальным коммуникационным узлом (можно представить себе виртуальный разъем или ящик для приема/отправки писем), ведущим от объекта во внешний мир и наоборот. При не прямой адресации сами промежуточные объекты для организации взаимодействия процессов также именуются сокетами.

Поскольку уровень Internet семейства протоколов TCP/IP умеет доставлять информацию только от компьютера к компьютеру, данные, полученные с его помощью, должны содержать тип использованного протокола транспортного уровня и локальные адреса отправителя и получателя. И протокол TCP, и протокол UDP используют не прямую адресацию.

Для того чтобы избежать путаницы, в дальнейшем в лабораторной работе термин «сокет» будет употребляться только для

обозначения самих промежуточных объектов, а полные адреса таких объектов будут называться адресами сокетов.

Для каждого транспортного протокола в стеке TCP/IP существуют собственные сокет: UDP сокет и TCP сокет, имеющие различные адресные пространства своих локальных адресов – портов. В семействе протоколов TCP/IP адресные пространства портов представляют собой положительные значения целого 16-битового числа. Поэтому, говоря о локальном адресе сокета, часто будет использоваться термин "номер порта". Из различия адресных пространств портов следует, что порт 1111 TCP – это совсем не тот же самый локальный адрес, что и порт 1111 UDP.

Иерархическая система адресации, используемая в семействе протоколов TCP/IP включает в себя несколько уровней:

Физический пакет данных, передаваемый по сети, содержит физические адреса узлов сети (MAC-адреса) с указанием на то, какой протокол уровня Internet должен использоваться для обработки передаваемых данных (поскольку пользователя интересуют только данные, доставляемые затем на уровень приложений/процессов, то для него это всегда IP).

IP-пакет данных содержит 32-битовые IP-адреса компьютера-отправителя и компьютера-получателя и указание на то, какой вышележащий протокол (TCP, UDP или еще что-нибудь) должен использоваться для их дальнейшей обработки.

Служебная информация транспортных протоколов (UDP-заголовок к данным и TCP-заголовок к данным) должна содержать 16-битовые номера портов для сокета отправителя и сокета получателя.

Добавление необходимой информации к данным при переходе от верхних уровней семейства протоколов к нижним принято называть английским словом encapsulation (дословно: герметизация). На рисунке 10.5 приведена схема encapsulation при использовании протокола UDP на сети Ethernet.

Поскольку между MAC-адресами и IP-адресами существует взаимно однозначное соответствие, известное семейству протоколов TCP/IP, то фактически для полного задания адреса доставки и адреса отправления, необходимых для установления двусторонней связи, нужно указать пять параметров:

<транспортный протокол, IP-адрес отправителя, порт отправителя, IP-адрес получателя, порт получателя>

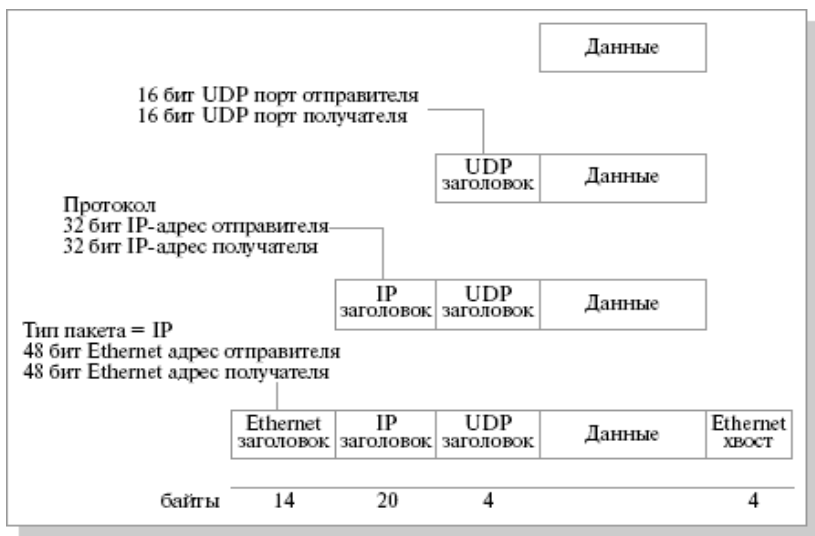


Рис. 10.5. Encapsulation для UDP-протокола на сети Ethernet

10.6 Использование модели клиент-сервер для взаимодействия удаленных процессов

Модель клиент-сервер, изначально предполагающая неравноправность взаимодействующих процессов, наиболее часто используется для организации сетевых приложений. Основные отличия процессов клиента и сервера применительно к удаленному взаимодействию:

- Сервер, как правило, работает постоянно, на всем протяжении жизни приложения, а клиенты могут работать эпизодически.
- Сервер ждет запроса от клиентов, инициатором же взаимодействия выступает клиент.
- Как правило, клиент обращается к одному серверу за раз, в то время как к серверу могут одновременно поступить запросы от нескольких клиентов.
- Клиент должен знать полный адрес сервера (его локальную и удаленную части) перед началом организации запроса (до начала

общения), в то время как сервер может получить информацию о полном адресе клиента из пришедшего запроса (после начала общения).

- И клиент, и сервер должны использовать один и тот же протокол транспортного уровня.

Неравноправность процессов в модели клиент-сервер накладывает свой отпечаток на программный интерфейс, используемый между уровнем приложений/процессов и транспортным уровнем.

Поступающие запросы сервер может обрабатывать последовательно – запрос за запросом – или параллельно, запуская для обработки каждого из них свой процесс или thread. Как правило, серверы, ориентированные на связь клиент-сервер с помощью установки логического соединения (TCP-протокол), ведут обработку запросов параллельно, а серверы, ориентированные на связь клиент-сервер без установления соединения (UDP-протокол), обрабатывают запросы последовательно.

10.7 Организация связи между удаленными процессами с помощью датаграмм

Наиболее простой для взаимодействия удаленных процессов является схема организации общения клиента и сервера с помощью датаграмм, т. е. использование протокола UDP.

С точки зрения обычного человека общение процессов посредством датаграмм напоминает общение людей в письмах. Каждое письмо представляет собой законченное сообщение, содержащее адрес получателя, адрес отправителя и указания, кто написал письмо и кто должен его получить. Письма могут теряться, доставляться в неправильном порядке, быть поврежденными в дороге и т.д.

Что в первую очередь должен сделать человек, проживающий в отдаленной местности, чтобы принимать и отправлять письма? Он должен изготовить почтовый ящик, который одновременно будет служить и для приема корреспонденции, и для ее отправки. Пришедшие письма почтальон будет помещать в этот ящик и забирать из него письма, подготовленные к отправке.

Изготовленный почтовый ящик нужно где-то прикрепить. Это может быть парадная дверь дома или вход со двора, изгородь, столб, дерево и т. п. Потенциально может быть изготовлено несколько почтовых ящиков и размещено в разных местах с тем, чтобы письма от различных адресатов прибывали в различные ящики. Этим ящикам будут соответствовать разные адреса: «Иванову, почтовый ящик на конюшне», «Иванову, почтовый ящик, что на дубе».

После закрепления ящика мы готовы к обмену корреспонденцией. Человек-клиент пишет письмо с запросом по заранее известному ему адресу человека-сервера и ждет получения ответного письма. После получения ответа он читает его и перерабатывает полученную информацию.

Человек-сервер изначально находится в состоянии ожидания запроса. Получив письмо, он читает текст запроса и определяет адрес отправителя. После обработки запроса он пишет ответ и отправляет его по обратному адресу, после чего начинает ждать следующего запроса.

Все эти модельные действия имеют аналоги при общении удаленных процессов по протоколу UDP.

Процесс-сервер должен сначала совершить подготовительные действия: создать UDP-сокеты (изготовить почтовый ящик) и связать его с определенным номером порта и IP-адресом сетевого интерфейса (прикрепить почтовый ящик в определенном месте) – настроить адрес сокета. При этом сокет может быть привязан к конкретному сетевому интерфейсу (к конюшне, к дубу) или к компьютеру в целом, то есть в полном адресе сокета может быть либо указан IP-адрес конкретного сетевого интерфейса, либо дано указание операционной системе, что информация может поступить через любой сетевой интерфейс, имеющийся в наличии. После настройки адреса сокета операционная система начинает принимать сообщения, пришедшие на этот адрес и складывать их в сокет. Сервер дожидается поступления сообщения, читает его, определяет, от кого оно поступило и через какой сетевой интерфейс, обрабатывает полученную информацию и отправляет результат по обратному адресу. После чего процесс готов к приему новой информации от того же или другого клиента.

Процесс-клиент должен сначала совершить те же самые подготовительные действия: создать сокет и настроить его адрес. Затем он передает сообщение, указав, кому оно предназначено (IP-

адрес сетевого интерфейса и номер порта сервера), ожидает от него ответа и продолжает свою деятельность.

Схематично эти действия выглядят так, как показано на рисунке 10.6. Каждому из них соответствует определенный системный вызов. Названия вызовов написаны справа от блоков соответствующих действий. Создание сокета производится с помощью системного вызова **socket()**. Для привязки созданного сокета к IP-адресу и номеру порта (настройка адреса) служит системный вызов **bind()**. Ожиданию получения информации, ее чтению и, при необходимости, определению адреса отправителя соответствует системный вызов **recvfrom()**. За отправку датаграммы отвечает системный вызов **sendto()**.

10.8 Сетевой порядок байт. Функции **htons()**, **htonl()**, **ntohs()**, **ntohl()**

Передача от одного вычислительного комплекса к другому символьной информации, как правило (когда один символ занимает один байт), не вызывает проблем. Однако для числовой информации ситуация усложняется.

Как известно, порядок байт в целых числах, представление которых занимает более одного байта, может быть для различных компьютеров неодинаковым. Есть вычислительные системы, в которых старший байт числа имеет меньший адрес, чем младший байт (**big-endian byte order**), а есть вычислительные системы, в которых старший байт числа имеет больший адрес, чем младший байт (**little-endian byte order**). При передаче целой числовой информации от машины, имеющей один порядок байт, к машине с другим порядком байт мы можем неправильно истолковать принятую информацию. Для того чтобы этого не произошло, было введено понятие сетевого порядка байт, т. е. порядка байт, в котором должна представляться целая числовая информация в процессе передачи ее по сети (на самом деле – это **big-endian byte order**). Целые числовые данные из представления, принятого на компьютере-отправителе, переводятся пользовательским процессом в сетевой порядок байт, в таком виде путешествуют по сети и переводятся в нужный порядок байт на машине-получателе процессом, которому они предназначены. Для перевода целых чисел из машинного представления в сетевое и обратно используется четыре функции: **htons()**, **htonl()**, **ntohs()**, **ntohl()**.

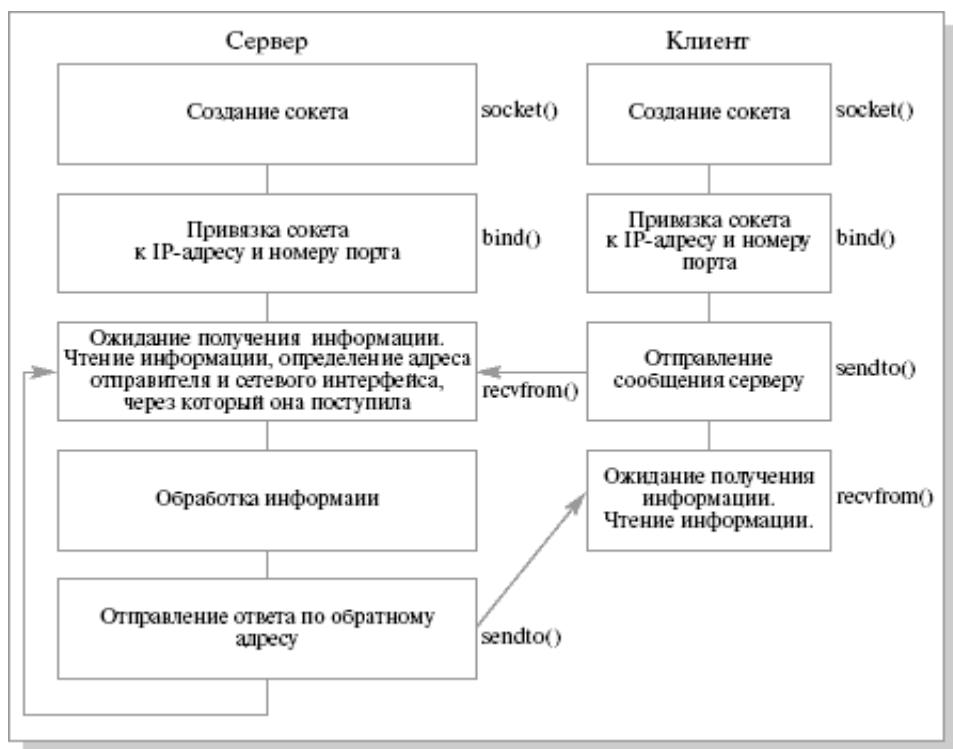


Рис. 10.6. Схема взаимодействия клиента и сервера для протокола UDP

Функции преобразования порядка байт

Прототипы функций

```
#include <netinet/in.h>
```

```
unsigned long int htonl(
    unsigned long int hostlong);
unsigned short int htons(
    unsigned short int hostshort);
unsigned long int ntohl(
    unsigned long int netlong);
unsigned short int ntohs(
    unsigned short int netshort);
```

Описание функций

- Функция **htonl** осуществляет перевод целого длинного числа из порядка байт, принятого на компьютере, в сетевой порядок байт.
- Функция **htons** осуществляет перевод целого короткого числа из порядка байт, принятого на компьютере, в сетевой порядок байт.
- Функция **ntohl** осуществляет перевод целого длинного числа из сетевого порядка байт в порядок байт, принятый на компьютере.
- Функция **ntohs** осуществляет перевод целого короткого числа из сетевого порядка байт в порядок байт, принятый на компьютере.

В архитектуре компьютеров i80x86 принят порядок байт, при котором младшие байты целого числа имеют младшие адреса. При сетевом порядке байт, принятом в Internet, младшие адреса имеют старшие байты числа. Параметр у них – значение, которое необходимо конвертировать. Возвращаемое значение – то, что получается в результате конвертации. Направление конвертации определяется порядком букв h (host) и n (network) в названии функции, размер числа – последней буквой названия, то есть **htons** – это **host to network short**, **ntohl** – **network to host long**.

Для чисел с плавающей точкой все обстоит гораздо хуже. На разных машинах могут различаться не только порядок байт, но и форма представления такого числа. Простых функций для их корректной передачи по сети не существует. Если требуется обмениваться действительными данными, то либо это нужно делать на гомогенной сети, состоящей из одинаковых компьютеров, либо использовать символьные и целые данные для передачи действительных значений.

10.9 Функции преобразования IP-адресов `inet_ntoa()`, `inet_aton()`

Функция **inet_aton()** переводит символьный IP-адрес в числовое представление в сетевом порядке байт.

Функция возвращает **1**, если в символьном виде записан правильный IP-адрес, и **0** в противном случае – для большинства системных вызовов и функций это нетипичная ситуация.

Обратите внимание на использование указателя на структуру `struct in_addr` в качестве одного из параметров данной функции. Эта структура используется для хранения IP-адресов в сетевом порядке байт. То, что используется структура, состоящая из одной переменной, а не сама 32-битовая переменная, сложилось исторически, и авторы в этом не виноваты.

Для обратного преобразования применяется функция `inet_ntoa()`.

Функции преобразования IP-адресов

Прототипы функций

```
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
int inet_aton(const char *strptr,
              struct in_addr *addrptr);
char *inet_ntoa(struct in_addr *addrptr);
```

Описание функций

Функция `inet_aton` переводит символьный IP-адрес, расположенный по указателю `strptr`, в числовое представление в сетевом порядке байт и заносит его в структуру, расположенную по адресу `addrptr`. Функция возвращает значение `1`, если в строке записан правильный IP-адрес, и значение `0` в противном случае. Структура типа `struct in_addr` используется для хранения IP-адресов в сетевом порядке байт и выглядит так:

```
struct in_addr {
    in_addr_t s_addr;
};
```

То, что используется адрес такой структуры, а не просто адрес переменной типа `in_addr_t`, сложилось исторически.

Функция `inet_ntoa` применяется для обратного преобразования. Числовое представление адреса в сетевом порядке байт должно быть занесено в структуру типа `struct in_addr`, адрес которой `addrptr` передается функции как аргумент. Функция возвращает указатель на

строку, содержащую символьное представление адреса. Эта строка располагается в статическом буфере, при последующих вызовах ее новое содержимое заменяет старое содержимое.

Функция **bzero**

Прототип функции

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

```
void bzero(void *addr, int n);
```

Описание функции

Функция **bzero** заполняет первые **n** байт, начиная с адреса **addr**, нулевыми значениями. Функция ничего не возвращает.

10.10 Создание сокета. Системный вызов **socket()**

Для создания сокета в операционной системе служит системный вызов **socket()**. Для транспортных протоколов семейства **TCP/IP** существует два вида сокетов: **UDP**-сокет – сокет для работы с датаграммами, и **TCP** сокет – потоковый сокет. Однако понятие сокета не ограничивается рамками только этого семейства протоколов. Рассматриваемый интерфейс сетевых системных вызовов (**socket()**, **bind()**, **recvfrom()**, **sendto()** и т. д.) в операционной системе UNIX может применяться и для других стеков протоколов (и для протоколов, лежащих ниже транспортного уровня). При создании сокета необходимо точно специфицировать его тип. Эта спецификация производится с помощью трех параметров вызова **socket()**. Первый параметр указывает, к какому семейству протоколов относится создаваемый сокет, а второй и третий параметры определяют конкретный протокол внутри данного семейства. Вторым параметром служит для задания вида интерфейса работы с сокетом – будет это потоковый сокет, сокет для работы с датаграммами или какой-либо иной. Третий параметр указывает протокол для заданного типа интерфейса. В стеке протоколов **TCP/IP** существует только один протокол для потоковых сокетов – **TCP** и только один протокол для датаграммных сокетов – **UDP**, поэтому для транспортных протоколов **TCP/IP** третий параметр игнорируется.

В других стеках протоколов может быть несколько протоколов с одинаковым видом интерфейса, например, датаграммных, различающихся по степени надежности.

Для транспортных протоколов TCP/IP в качестве первого параметра указываем предопределенную константу **AF_INET** (**Address family – Internet**) или ее синоним **PF_INET** (**Protocol family – Internet**).

Второй параметр будет принимать предопределенные значения **SOCK_STREAM** для потоковых сокетов и **SOCK_DGRAM** – для датаграммных.

Поскольку третий параметр в данном случае не учитывается, в него мы будем подставлять значение **0**. Ссылка на информацию о созданном соquete помещается в таблицу открытых файлов процесса подобно тому, как это делалось для `pip`'ов и `FIFO`. Системный вызов возвращает пользователю файловый дескриптор, соответствующий заполненному элементу таблицы, который далее мы будем называть дескриптором сокета. Такой способ хранения информации о соquete позволяет, во-первых, процессам-детям наследовать ее от процессородителей, а во-вторых, использовать для сокетов часть системных вызовов, которые уже знакомы нам по работе с `pip`'ами и `FIFO`: `close()`, `read()`, `write()`.

Системный вызов для создания сокета

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type,
           int protocol)
```

Описание системного вызова

Системный вызов **socket** служит для создания виртуального коммуникационного узла в операционной системе. Параметр `domain` определяет семейство протоколов, в рамках которого будет осуществляться передача информации. Мы рассмотрим только два таких семейства из нескольких существующих. Для них имеются предопределенные значения параметра:

PF_INET – для семейства протоколов TCP/IP ;

PF_UNIX – для семейства внутренних протоколов UNIX, иначе называемого еще UNIX domain.

Параметр **type** определяет семантику обмена информацией: будет ли осуществляться связь через сообщения (datagrams), с помощью установления виртуального соединения или еще каким-либо способом. Будем пользоваться только двумя способами обмена информацией с предопределенными значениями для параметра type:

SOCK_STREAM – для связи с помощью установления виртуального соединения;

SOCK_DGRAM – для обмена информацией через сообщения.

Параметр **protocol** специфицирует конкретный протокол для выбранного семейства протоколов и способа обмена информацией. Он имеет значение только в том случае, когда таких протоколов существует несколько

Возвращаемое значение

В случае успешного завершения системный вызов возвращает файловый дескриптор (значение большее или равное 0), который будет использоваться как ссылка на созданный коммуникационный узел при всех дальнейших сетевых вызовах. При возникновении какой-либо ошибки возвращается отрицательное значение.

10.11 Адреса сокетов. Настройка адреса сокета. Системный вызов bind()

Когда сокет создан, необходимо настроить его адрес. Для этого используется системный вызов **bind()**. Первый параметр вызова должен содержать дескриптор сокета, для которого производится настройка адреса. Второй и третий параметры задают этот адрес.

Во втором параметре должен быть указатель на структуру **struct sockaddr**, содержащую удаленную и локальные части полного адреса.

Указатели типа **struct sockaddr *** встречаются во многих сетевых системных вызовах; они используются для передачи информации о том, к какому адресу привязан или должен быть привязан сокет. Рассмотрим этот тип данных подробнее. Структура **struct sockaddr** описана в файле **<sys/socket.h>** следующим образом:

```

struct sockaddr
{
    short sa_family;
    char sa_data[14];
};

```

акой состав структуры обусловлен тем, что сетевые системные вызовы могут применяться для различных семейств протоколов, которые по-разному определяют адресные пространства для удаленных и локальных адресов сокета. По сути дела, этот тип данных представляет собой лишь общий шаблон для передачи системным вызовам структур данных, специфических для каждого семейства протоколов. Общим элементом этих структур остается только поле `short sa_family` (которое в разных структурах, естественно, может иметь разные имена, важно лишь, чтобы все они были одного типа и были первыми элементами своих структур) для описания семейства протоколов. Содержимое этого поля системный вызов анализирует для точного определения состава поступившей информации.

Для работы с семейством протоколов TCP/IP будем использовать адрес сокета следующего вида, описанного в файле `<netinet/in.h>`:

```

struct sockaddr_in
{
    short sin_family;
    /* Избранное семейство протоколов – всегда AF_INET */
    unsigned short sin_port;
    /* 16-битовый номер порта в сетевом порядке байт */
    struct in_addr sin_addr;
    /* Адрес сетевого интерфейса */
    char sin_zero[8];
    /* Это поле не используется, но должно всегда быть заполнено
    нулями */
};

```

Первый элемент структуры – `sin_family` задает семейство протоколов. В него необходимо занести предопределенную константу `AF_INET`.

Удаленная часть полного адреса – IP-адрес – содержится в структуре типа **struct in_addr**.

Для указания номера порта предназначен элемент структуры **sin_port**, в котором номер порта должен храниться в сетевом порядке байт. Существует два варианта задания номера порта: фиксированный порт по желанию пользователя и порт, который произвольно назначает операционная система. Первый вариант требует указания в качестве номера порта положительного заранее известного числа и для протокола UDP обычно используется при настройке адресов сокетов и при передаче информации с помощью системного вызова `sendto()` (следующий раздел). Второй вариант требует указания в качестве номера порта значения 0. В этом случае операционная система сама привязывает сокет к свободному номеру порта. Этот способ обычно используется при настройке сокетов программ клиентов, когда заранее точно знать номер порта программисту необязательно.

Какой номер порта может задействовать пользователь при фиксированной настройке? Номера портов с 1 по 1023 могут назначать сокетам только процессы, работающие с привилегиями системного администратора. Как правило, эти номера закреплены за системными сетевыми службами независимо от вида используемой операционной системы, для того чтобы пользовательские клиентские программы могли запрашивать обслуживание всегда по одним и тем же локальным адресам. Существует также ряд широко применяемых сетевых программ, которые запускают процессы с полномочиями обычных пользователей (например, X-Windows). Для таких программ корпорацией Internet по присвоению имен и номеров (ICANN) выделяется диапазон адресов с 1024 по 49151, который нежелательно использовать во избежание возможных конфликтов. Номера портов с 49152 по 65535 предназначены для процессов обычных пользователей. Во всех наших примерах при фиксированном задании номера порта у сервера мы будем использовать номер 51000.

IP-адрес при настройке также может быть определен двумя способами. Он может быть привязан к конкретному сетевому интерфейсу (т. е. сетевой плате), заставляя операционную систему принимать/передавать информацию только через этот сетевой интерфейс, а может быть привязан и ко всей вычислительной системе в целом (информация может быть получена/отослана через любой сетевой интерфейс).

В первом случае в качестве значения поля структуры **sin_addr.s_addr** используется числовое значение IP-адреса конкретного сетевого интерфейса в сетевом порядке байт. Во втором случае это значение должно быть равно значению предопределенной константы **INADDR_ANY**, приведенному к сетевому порядку байт.

Третий параметр системного вызова **bind()** должен содержать фактическую длину структуры, адрес которой передается в качестве второго параметра. Эта длина меняется в зависимости от семейства протоколов и даже различается в пределах одного семейства протоколов. Размер структуры, содержащей адрес сокета, для семейства протоколов TCP/IP может быть определен как **sizeof(struct sockaddr_in)**.

Системный вызов для привязки сокета к конкретному адресу

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockd,
        struct sockaddr *my_addr,
        int addrlen);
```

Описание системного вызова

Системный вызов **bind** служит для привязки созданного сокета к определенному полному адресу вычислительной сети.

Параметр **sockd** является дескриптором созданного ранее коммуникационного узла, т. е. значением, которое вернул системный вызов **socket()**.

Параметр **my_addr** представляет собой адрес структуры, содержащей информацию о том, куда именно мы хотим привязать наш сокет – то, что принято называть адресом сокета. Он имеет тип указателя на структуру-шаблон **struct sockaddr**, которая должна быть конкретизирована в зависимости от используемого семейства протоколов и заполнена перед вызовом.

Параметр **addrlen** должен содержать фактическую длину структуры, адрес которой передается в качестве второго параметра. Эта длина в разных семействах протоколов и даже в пределах одного семейства протоколов может быть различной (например, для UNIX Domain).

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и отрицательное значение – в случае ошибки.

10.12 Системные вызовы `sendto()` и `recvfrom()`

Для отправки датаграмм применяется системный вызов `sendto()`. В число параметров этого вызова входят:

- дескриптор сокета, через который отсылается датаграмма;
- адрес области памяти, где лежат данные, которые должны составить содержательную часть датаграммы, и их длина;
- флаги, определяющие поведение системного вызова (в нашем случае они всегда будут иметь значение 0);
- указатель на структуру, содержащую адрес сокета получателя, и ее фактическая длина.

Системный вызов возвращает отрицательное значение при возникновении ошибки и количество реально отосланных байт при нормальной работе. Нормальное завершение системного вызова не означает, что датаграмма уже покинула ваш компьютер! Датаграмма сначала помещается в системный сетевой буфер, а ее реальная отправка может произойти после возврата из системного вызова. Вызов `sendto()` может блокироваться, если в сетевом буфере не хватает места для датаграммы.

Для чтения принятых датаграмм и определения адреса получателя (при необходимости) служит системный вызов `recvfrom()`. В число параметров этого вызова входят:

- Дескриптор сокета, через который принимается датаграмма.
- Адрес области памяти, куда следует положить данные, составляющие содержательную часть датаграммы.
- Максимальная длина, допустимая для датаграммы. Если количество данных датаграммы превышает заданную максимальную длину, то вызов по умолчанию рассматривает это как ошибочную ситуацию.
- Флаги, определяющие поведение системного вызова (в нашем случае они будут полагаться равными 0).
- Указатель на структуру, в которую при необходимости может быть занесен адрес сокета отправителя. Если этот адрес не

требуется, то можно указать значение **NULL**.

- Указатель на переменную, содержащую максимально возможную длину адреса отправителя.

После возвращения из системного вызова в нее будет занесена фактическая длина структуры, содержащей адрес отправителя. Если предыдущий параметр имеет значение **NULL**, то и этот параметр может иметь значение **NULL**.

Системный вызов **recvfrom()** по умолчанию блокируется, если отсутствуют принятые датаграммы, до тех пор, пока датаграмма не появится. При возникновении ошибки он возвращает отрицательное значение, при нормальной работе – длину принятой датаграммы.

Системные вызовы **sendto** и **recvfrom**

Прототипы системных вызовов

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

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

```
int sendto(int sockd, char *buff,  
           int nbytes, int flags,  
           struct sockaddr *to, int addrlen);  
int recvfrom(int sockd, char *buff,  
            int nbytes, int flags,  
            struct sockaddr *from, int *addrlen);
```

Описание системных вызовов

Системный вызов **sendto** предназначен для отправки датаграмм. Системный вызов **recvfrom** предназначен для чтения пришедших датаграмм и определения адреса отправителя. По умолчанию при отсутствии пришедших датаграмм вызов **recvfrom** блокируется до тех пор, пока не появится датаграмма. Вызов **sendto** может блокироваться при отсутствии места под датаграмму в сетевом буфере. Данное описание не является полным описанием системных вызовов, а предназначено только для использования в нашем курсе. За полной информацией обращайтесь к UNIX Manual.

Параметр **sockd** является дескриптором созданного ранее сокета, т. е. значением, возвращенным системным вызовом **socket()**, через который будет отсылаться или получаться информация.

Параметр **buff** представляет собой адрес области памяти, начиная с которого будет браться информация для передачи или размещаться принятая информация.

Параметр **nbytes** для системного вызова **sendto** определяет количество байт, которое должно быть передано, начиная с адреса памяти **buff**. Параметр **nbytes** для системного вызова **recvfrom** определяет максимальное количество байт, которое может быть размещено в приемном буфере, начиная с адреса **buff**.

Параметр **to** для системного вызова **sendto** определяет ссылку на структуру, содержащую адрес сокета получателя информации, которая должна быть заполнена перед вызовом. Если параметр **from** для системного вызова **recvfrom** не равен **NULL**, то для случая установления связи через пакеты данных он определяет ссылку на структуру, в которую будет занесен адрес сокета отправителя информации после завершения вызова. В этом случае перед вызовом эту структуру необходимо обнулить.

Параметр **addrlen** для системного вызова **sendto** должен содержать фактическую длину структуры, адрес которой передается в качестве параметра **to**. Для системного вызова **recvfrom** параметр **addrlen** является ссылкой на переменную, в которую будет занесена фактическая длина структуры адреса сокета отправителя, если это определено параметром **from**. Заметим, что перед вызовом этот параметр должен указывать на переменную, содержащую максимально допустимое значение такой длины. Если параметр **from** имеет значение **NULL**, то и параметр **addrlen** может иметь значение **NULL**.

Параметр **flags** определяет режимы использования системных вызовов. Рассматривать его применение мы в данном курсе не будем, и поэтому берем значение этого параметра равным 0.

Возвращаемое значение

В случае успешного завершения системный вызов возвращает количество реально отосланных или принятых байт. При возникновении какой-либо ошибки возвращается отрицательное значение.

10.13 Определение IP-адресов для вычислительного комплекса

Для определения IP-адресов на компьютере можно воспользоваться утилитой **/sbin/ifconfig**. Эта утилита выдает всю информацию о сетевых интерфейсах, сконфигурированных в вычислительной системе.

Пример выдачи утилиты показан ниже:

```
eth0  Link encap:Ethernet HWaddr 00:90:27:A7:1B:FE
      inet addr:192.168.253.12 Bcast:192.168.253.255
      Mask:255.255.255.0
      UP BROADCAST NOTRAILERS RUNNING MULTICAST MTU:1500
      Metric:1 RX packets:122556059 errors:0 dropped:0
      overruns:0 frame:0 TX packets:116085111 errors:0
      dropped:0 overruns:0 carrier:0 collisions:0
      txqueuelen:100 RX bytes:2240402748 (2136.6 Mb)
      TX bytes:3057496950 (2915.8 Mb) Interrupt:10
      Base address:0x1000
lo    Link encap:Local Loopback
      inet addr:127.0.0.1 Mask:255.0.0.0
      UP LOOPBACK RUNNING MTU:16436 Metric:1
      RX packets:403 errors:0 dropped:0 overruns:0 frame:0
      TX packets:403 errors:0 dropped:0 overruns:0
      carrier:0 collisions:0 txqueuelen:0
      RX bytes:39932 (38.9 Kb) TX bytes:39932 (38.9 Kb)
```

Сетевой интерфейс eth0 использует протокол Ethernet. Физический 48-битовый адрес, зашитый в сетевой карте, – 00:90:27:A7:1B:FE. Его IP-адрес – 192.168.253.12.

Сетевой интерфейс lo не относится ни к какой сетевой карте. Это так называемый локальный интерфейс, который через общую память эмулирует работу сетевой карты для взаимодействия процессов, находящихся на одной машине по полным сетевым адресам. Наличие этого интерфейса позволяет отлаживать сетевые программы на машинах, не имеющих сетевых карт. Его IP-адрес обычно одинаков на всех компьютерах – 127.0.0.1.

Рассмотрим простой пример программы 10-01. Эта программа является UDP-клиентом для стандартного системного сервиса echo.

Стандартный сервис принимает от клиента текстовую датаграмму и, не изменяя ее, отправляет обратно. За сервисом зарезервирован номер порта 7. Для правильного запуска программы необходимо указать символьный IP-адрес сетевого интерфейса компьютера, к сервису которого нужно обратиться, в качестве аргумента командной строки, например:

a.out 192.168.253.12

Программа 10-01. Простой пример UDP клиента для сервиса echo

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <iostream>
using namespace std;
int main(int argc, char **argv)
{
    int sockfd; /* Дескриптор сокета */
    int n, len; /* Переменные для различных длин и количества символов */
    char sendline[1000], recvline[1000]; /* Массивы для отсылаемой и
    принятой строки */
    struct sockaddr_in servaddr, cliaddr; /* Структуры для адресов
    сервера и клиента */
    /* Сначала проверяем наличие второго аргумента в командной
    строке. При его отсутствии прекращаем работу */
    if(argc != 2){
        cout<<"Usage: a.out <IP address>"<<endl;
        exit(1);
    }
    /* Создание UDP сокета */
    if((sockfd = socket(PF_INET, SOCK_DGRAM, 0)) < 0){
        perror(NULL); /* Печать сообщения об ошибке */
```

```

    exit(1);
}
/* Заполнение структуры для адреса клиента: семейство протоколов
TCP/IP, сетевой интерфейс – любой, номер порта по усмотрению
операционной системы. Поскольку в структуре содержится
дополнительное ненужное поле, которое должно быть нулевым, перед
заполнением обнуляем ее всю */
bzero(&cliaddr, sizeof(cliaddr));
cliaddr.sin_family = AF_INET;
cliaddr.sin_port = htons(0);
cliaddr.sin_addr.s_addr = htonl(INADDR_ANY);
/* Настройка адреса сокета */
if(bind(sockfd, (struct sockaddr *) &cliaddr, sizeof(cliaddr)) < 0){
    perror(NULL);
    close(sockfd); /* По окончании работы закрываем дескриптор
сокета */
    exit(1);
}
/* Заполнение структуры для адреса сервера: семейство протоколов
TCP/IP, сетевой интерфейс – из аргумента командной строки, номер
порта 7. Поскольку в структуре содержится дополнительное ненужное
поле, которое должно быть нулевым, перед заполнением обнуляем ее
всю */
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(7);
if(inet_aton(argv[1], &servaddr.sin_addr) == 0){
    cout<<"Invalid IP address"<<endl;
    close(sockfd); /* По окончании работы закрываем дескриптор
сокета */
    exit(1);
}
/* Ввод строки, которую отошлем серверу */
cout<<"String => "<<endl;
fgets(sendline, 1000, stdin);
/* Отсылка датаграммы */
if(sendto(sockfd, sendline, strlen(sendline)+1, 0, (struct sockaddr *)
&servaddr, sizeof(servaddr)) < 0){

```

```

    perror(NULL);
    close(sockfd);
    exit(1);
}
/* Ожидание ответа и чтение его. Максимальная допустимая длина
датограммы – 1000 символов, адрес отправителя не нужен */
if((n = recvfrom(sockfd, recvline, 1000, 0, (struct sockaddr *) NULL,
NULL)) < 0){
    perror(NULL);
    close(sockfd);
    exit(1);
}
/* Печатаем пришедший ответ и закрываем сокет */
cout<< recvline << endl;
close(sockfd);
return 0;
}

```

Листинг 10-01. Простой пример UDP клиента для сервиса echo

Поскольку UDP-сервер использует те же самые системные вызовы, что и UDP-клиент, можно сразу приступить к рассмотрению примера UDP-сервера (программа 10-02) для сервиса echo.

Программа 10-02. Простой пример UDP-сервера для сервиса echo

```

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <iostream>
using namespace std;
int main()
{
    int sockfd; /* Дескриптор сокета */
    int cliilen, n; /* Переменные для различных длин и количества
символов */

```

```

char line[1000]; /* Массив для принятой и отсылаемой строки */
struct sockaddr_in servaddr, cliaddr; /* Структуры для адресов
сервера и клиента */
/* Заполнение структуры для адреса сервера: семейство
протоколов TCP/IP, сетевой интерфейс – любой, номер порта 51000.
Поскольку в структуре содержится дополнительное ненужное поле,
которое должно быть нулевым, перед заполнением обнуляем ее всю
*/
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(51000);
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
/* Создание UDP сокета */
if((sockfd = socket(PF_INET, SOCK_DGRAM, 0)) < 0){
    perror(NULL); /* Печать сообщения об ошибке */
    exit(1);
}
/* Настройка адреса сокета */
if(bind(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0){
    perror(NULL);
    close(sockfd);
    exit(1);
}
while(1) {
    /* Основной цикл обслуживания */
    /* В переменную clilen заносим максимальную длину для ожидаемого
адреса клиента */
    clilen = sizeof(cliaddr);
    /* Ожидание прихода запроса от клиента и чтение его.
Максимальная допустимая длина датаграммы – 999 символов, адрес
отправителя помещаем в структуру cliaddr, его реальная длина будет
занесена в переменную clilen */
    if((n = recvfrom(sockfd, line, 999, 0, (struct sockaddr *) &cliaddr,
&clilen)) < 0){
        perror(NULL);
        close(sockfd);
        exit(1);
    }
}

```

```

/* Печать принятого текста на экране */
cout<< line<<endl;
/* Принятый текст отправляем обратно по адресу отправителя */
if(sendto(sockfd, line, strlen(line), 0, (struct sockaddr *) &cliaddr,
clilen) < 0){
    perror(NULL);
    close(sockfd);
    exit(1);
} /* Ожидание новой датаграммы*/
}
return 0;
}

```

Листинг 10-02. Простой пример UDP-сервера для сервиса echo

10.14 Организация связи между процессами с помощью установки логического соединения

Рассмотрим организацию взаимодействия процессов с помощью протокола ТСР, то есть при помощи создания логического соединения. Если взаимодействие процессов через датаграммы напоминает общение людей по переписке, то для протокола ТСР лучшей аналогией является общение людей по телефону. Какие действия должен выполнить клиент для того, чтобы связаться по телефону с сервером? Во-первых, необходимо приобрести телефон (создать сокет), во-вторых, подключить его к АТС – получить номер (настроить адрес сокета). Далее требуется позвонить серверу (установить логическое соединение). После установления соединения можно неоднократно обмениваться с сервером информацией (писать и читать из потока данных). По окончании взаимодействия нужно повесить трубку (закрыть сокет).

Первые действия сервера аналогичны действиям клиента. Он должен приобрести телефон и подключить его к АТС (создать сокет и настроить его адрес). А вот дальше поведение клиента и сервера различно. Представьте себе, что телефоны изначально продаются с выключенным звонком. Звонить по ним можно, а вот принять звонок – нет. Для того чтобы вы могли пообщаться, необходимо включить звонок. В терминах сокетов это означает, что ТСР-сокет по умолчанию создается в активном состоянии и предназначен не для

приема, а для установления соединения. Для того чтобы соединение принять, сокет требуется перевести в пассивное состояние.

Если два человека беседуют по телефону, то попытка других людей дозвониться до них окажется неудачной. Будет идти сигнал «занято», и соединение не установится. В то же время хотелось бы, чтобы клиент в такой ситуации не получал отказ в обслуживании, а ожидал своей очереди. Подобное наблюдается в различных телефонных справочных, когда вы слышите «Ждите, пожалуйста, ответа. Вам обязательно ответит оператор». Поэтому следующее действие сервера – это создание очереди для обслуживания клиентов. Далее сервер должен дожидаться установления соединения, прочитать информацию, переданную по линии связи, обработать ее и отправить полученный результат обратно. Обмен информацией может осуществляться неоднократно. Заметим, что сокет, находящийся в пассивном состоянии, не предназначен для операций приема и передачи информации. Для общения на сервере во время установления соединения автоматически создается новый потоковый сокет, через который и производится обмен данными с клиентами. По окончании общения сервер «кладет трубку» (закрывает этот новый сокет) и отправляется ждать очередного звонка.

Схематично эти действия выглядят так, как показано на рисунке 10.7. Как и в случае протокола UDP, отдельным действиям или их группам соответствуют системные вызовы, частично совпадающие с вызовами для протокола UDP. Их названия написаны справа от блоков соответствующих действий.

Для протокола TCP неравноправность процессов клиента и сервера видна особенно отчетливо в различии используемых системных вызовов. Для создания сокетов и там, и там по-прежнему используется системный вызов `socket()` . Затем наборы системных вызовов становятся различными.

Для привязки сервера к IP-адресу и номеру порта, как и в случае UDP-протокола, используется системный вызов `bind()`. Для процесса клиента эта привязка объединена с процессом установления соединения с сервером в новом системном вызове `connect()` и скрыта от глаз пользователя. Внутри этого вызова операционная система осуществляет настройку сокета на выбранный ею порт и на адрес любого сетевого интерфейса. Для перевода сокета на сервере в пассивное состояние и для создания очереди соединений служит

системный вызов `listen()`. Сервер ожидает соединения и получает информацию об адресе соединившегося с ним клиента с помощью системного вызова `accept()`. Поскольку установленное логическое соединение выглядит со стороны процессов как канал связи, позволяющий обмениваться данными с помощью потоковой модели, для передачи и чтения информации оба системных вызова используют уже известные нам системные вызовы `read()` и `write()`, а для завершения соединения – системный вызов `close()`. При работе с сокетами вызовы `read()` и `write()` обладают теми же особенностями поведения, что и при работе с `pip`'ами и `FIFO`.

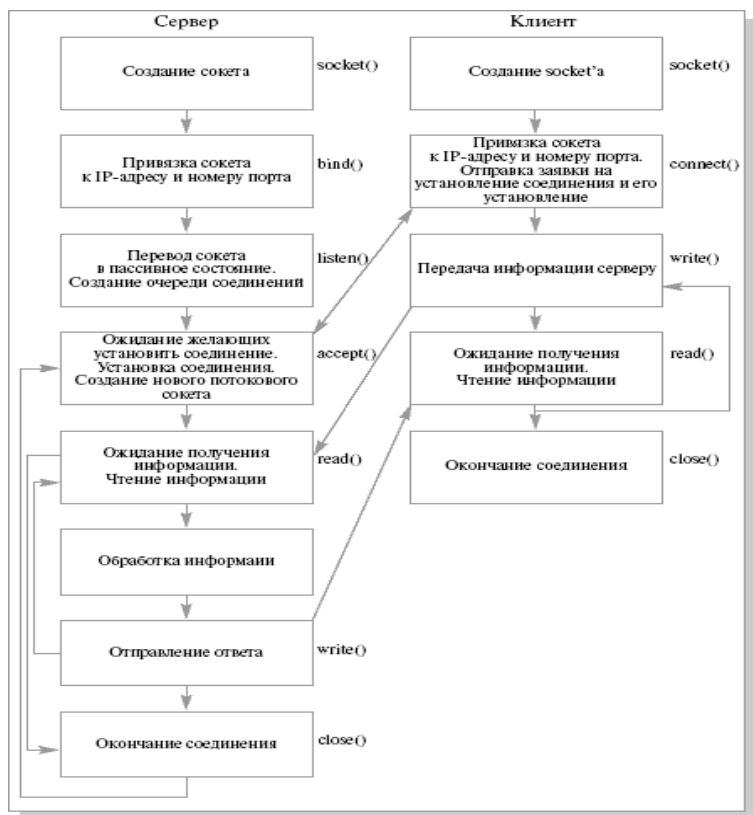


Рис. 10.7. Схема взаимодействия клиента и сервера для протокола TCP

10.15 Установление логического соединения. Системный вызов connect()

Среди системных вызовов со стороны клиента появляется только один новый – connect(). Системный вызов connect() при работе с TCP-сокетами служит для установления логического соединения со стороны клиента. Вызов connect() скрывает внутри себя настройку сокета на выбранный системой порт и произвольный сетевой интерфейс (вызов bind() с нулевым номером порта и IP-адресом **INADDR_ANY**). Вызов блокируется до тех пор, пока не будет установлено логическое соединение или пока не пройдет определенный промежуток времени, который может регулироваться системным администратором.

Для установления соединения необходимо задать три параметра: дескриптор активного сокета, через который будет устанавливаться соединение, полный адрес сокета сервера и его длину.

Системный вызов connect()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockd,
            struct sockaddr *servaddr,
            int addrlen);
```

Описание системного вызова

Системный вызов connect служит для организации связи клиента с сервером. Чаще всего он используется для установления логического соединения, хотя может быть применен и при связи с помощью датаграмм (connectionless). Данное описание не является полным описанием системного вызова, а предназначено только для использования в нашем курсе. Полную информацию можно найти в UNIX Manual.

Параметр **sockd** является дескриптором созданного ранее коммуникационного узла, т. е. значением, которое вернул системный вызов socket().

Параметр **servaddr** представляет собой адрес структуры, содержащей информацию о полном адресе сокета сервера. Он имеет тип указателя на структуру-шаблон `struct sockaddr`, которая должна быть конкретизирована в зависимости от используемого семейства протоколов и заполнена перед вызовом.

Параметр **addrlen** должен содержать фактическую длину структуры, адрес которой передается в качестве второго параметра. Эта длина меняется в зависимости от семейства протоколов и различается даже в пределах одного семейства протоколов (например, для UNIX Domain).

При установлении виртуального соединения системный вызов не возвращается до его установления или до истечения установленного в системе времени – `timeout`. При использовании его в `connectionless` связи вызов возвращается немедленно.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и отрицательное значение, если в процессе его выполнения возникла ошибка.

Рассмотрим пример – **программу 10-03**. Это простой TCP-клиент, обращающийся к стандартному системному сервису **echo**. Стандартный сервис принимает от клиента текстовую датаграмму и, не изменяя ее, отправляет обратно. За сервисом зарезервирован номер порта 7. *Заметим, что это порт 7 TCP – не путать с портом 7 UDP из примера в разделе «Пример программы UDP-клиента»!* Для правильного запуска программы необходимо указать символьный IP-адрес сетевого интерфейса компьютера, к сервису которого требуется обратиться, в качестве аргумента командной строки, например:

\$a.out 192.168.253.12

Для того чтобы подчеркнуть, что после установления логического соединения клиент и сервер могут обмениваться информацией неоднократно, клиент трижды запрашивает текст с экрана, отправляет его серверу и печатает полученный ответ.

Программа 10-03. Простой пример TCP-клиента для сервиса echo

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <iostream>
using namespace std;
void main(int argc, char **argv)
{
    int sockfd; /* Дескриптор сокета */
    int n; /* Количество переданных или прочитанных символов */
    int i; /* Счетчик цикла */
    char sendline[1000],recvline[1000]; /* Массивы для отсылаемой и
принятой строки */
    struct sockaddr_in servaddr; /* Структура для адреса сервера */
    /* Сначала проверка наличия второго аргумента в командной
строке. При его отсутствии – прекращение работы */
    if(argc != 2){
        cout<<"Usage: a.out <IP address>"<<endl;
        exit(1);
    }
    /* Обнуление символьных массивов */
    bzero(sendline,1000);
    bzero(recvline,1000);
    /* Создание TCP сокета */
    if((sockfd = socket(PF_INET, SOCK_STREAM, 0)) < 0){
        perror(NULL); /* Печать сообщения об ошибке */
        exit(1);
    }
    /* Заполнение структуры для адреса сервера: семейство
протоколов TCP/IP, сетевой интерфейс – из аргумента командной
строки, номер порта 7. Поскольку в структуре содержится
дополнительное ненужное поле, которое должно быть нулевым,
```

```

перед заполнением обнуляем ее всю */
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(51000);
if(inet_aton(argv[1], &servaddr.sin_addr) == 0){
    cout<<"Invalid IP address"<< endl;
    close(sockfd);
    exit(1);
}
/* Установка логического соединения через созданный сокет с
сокетом сервера, адрес которого занесли в структуру */
if(connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0){
    perror(NULL);
    close(sockfd);
    exit(1);
}
/* Три раза в цикле ввод строки с клавиатуры, отправка ее серверу
и чтение полученного ответа */
for(i=0; i<3; i++){
    cout<<"String => "<<endl;
    fflush(stdin);
    fgets(sendline, 1000, stdin);
    if( (n = write(sockfd, sendline, strlen(sendline)+1)) < 0){
        perror("Can't write\n");
        close(sockfd);
        exit(1);
    }
    if( (n = read(sockfd, recvline, 999)) < 0){
        perror("Can't read\n");
        close(sockfd);
        exit(1);
    }
    cout<<recvline<<endl;
}
/* Завершение соединения*/
close(sockfd);
}

```

Листинг 10-03. Простой пример TCP-клиента для сервиса echo

10.17 Как происходит установление виртуального соединения

Протокол TCP является надежным дуплексным протоколом. С точки зрения пользователя работа через протокол TCP выглядит как обмен информацией через поток данных. Внутри сетевых частей операционных систем поток данных отправителя нарезается на пакеты данных, которые, собственно, путешествуют по сети и на машине-получателе вновь собираются в выходной поток данных. В протоколе TCP используются приемы нумерации передаваемых пакетов и контроля порядка их получения, подтверждения о приеме пакета со стороны получателя и подсчет контрольных сумм по передаваемой информации. Для правильного порядка получения пакетов получатель должен знать начальный номер первого пакета отправителя. Поскольку связь является дуплексной и в роли отправителя пакетов данных могут выступать обе взаимодействующие стороны, они до передачи пакетов данных должны обменяться, по крайней мере, информацией об их начальных номерах. Согласование начальных номеров происходит по инициативе клиента при выполнении системного вызова `connect()`. Для такого согласования клиент посылает серверу специальный пакет информации, который принято называть SYN (от слова *synchronize* – синхронизировать). Он содержит как минимум начальный номер для пакетов данных, который будет использовать клиент. Сервер должен подтвердить получение пакета SYN от клиента и отправить ему свой пакет SYN с начальным номером для пакетов данных в виде единого пакета с сегментами SYN и ACK (от слова *acknowledgement* – подтверждение). В ответ клиент пакетом данных ACK должен подтвердить прием пакета данных от сервера.

Описанная выше процедура, получившая название трехэтапного рукопожатия (*three-way handshake*), схематично изображена на рисунке 10.8. При приеме на машине-сервере пакета SYN, направленного на пассивный (слушающий) сокет, сетевая часть операционной системы создает копию этого сокета – присоединенный сокет – для последующего общения, отмечая его как сокет с не полностью установленным соединением. После приема от клиента пакета ACK этот сокет переводится в состояние полностью установленного соединения, и тогда он готов к дальнейшей работе с использованием вызовов `read()` и `write()`.

10.18 Системный вызов listen()

Системный вызов **listen()** является первым из еще неизвестных нам вызовов, применяемым на ТСП-сервере. В его задачу входит перевод ТСП-сокета в пассивное (слушающее) состояние и создание очередей для порождаемых при установлении соединения присоединенных сокетов, находящихся в состоянии не полностью установленного соединения и полностью установленного соединения. Для этого вызов имеет два параметра: дескриптор ТСП-сокета и число, определяющее глубину создаваемых очередей.

Системный вызов listen()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/socket.h>
int listen(int sockd, int backlog);
```

Описание системного вызова

Системный вызов **listen** используется сервером, ориентированным на установление связи путем виртуального соединения, для перевода сокета в пассивный режим и установления глубины очереди для соединений.

Параметр **sockd** является дескриптором созданного ранее сокета, который должен быть переведен в пассивный режим, т. е. значением, которое вернул системный вызов **socket()**. Системный вызов **listen** требует предварительной настройки адреса сокета с помощью системного вызова **bind()**.

Параметр **backlog** определяет максимальный размер очередей для сокетов, находящихся в состояниях полностью и не полностью установленных соединений.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Последний параметр на разных UNIX-подобных операционных системах и даже на разных версиях одной и той же системы может

иметь различный смысл. Где-то это суммарная длина обеих очередей, где-то он относится к очереди не полностью установленных соединений (например, Linux до версии ядра 2.2) где-то – к очереди полностью установленных соединений (например, Linux, начиная с версии ядра 2.2), где-то – вообще игнорируется.

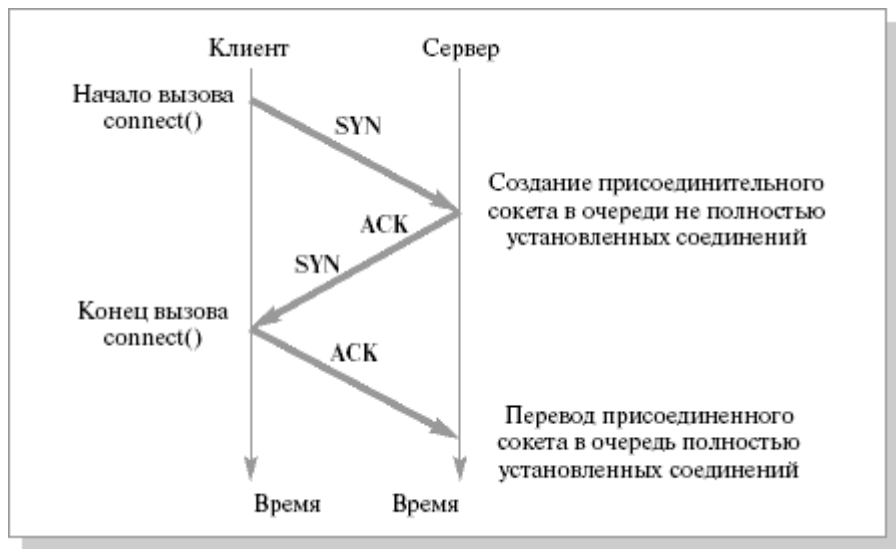


Рис. 10.8. Схема установления TCP соединения

10.19 Системный вызов accept()

Системный вызов **accept()** позволяет серверу получить информацию о полностью установленных соединениях. Если очередь полностью установленных соединений не пуста, то он возвращает дескриптор для первого присоединенного сокета в этой очереди, одновременно удаляя его из очереди. Если очередь пуста, то вызов ожидает появления полностью установленного соединения. Системный вызов также позволяет серверу узнать полный адрес клиента, установившего соединение. У вызова есть три параметра: дескриптор слушающего сокета, через который ожидается установление соединения; указатель на структуру, в которую при необходимости будет занесен полный адрес сокета клиента,

установившего соединение; указатель на целую переменную, содержащую максимально допустимую длину этого адреса. Как и в случае вызова `recvfrom()`, последний параметр является модернизируемым, а если нас не интересует, кто с нами соединился, то вместо второго и третьего параметров можно указать значение `NULL`.

Системный вызов `accept()`

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockd,
           struct sockaddr *cliaddr,
           int *clilen);
```

Описание системного вызова

Системный вызов `accept` используется сервером, ориентированным на установление связи путем виртуального соединения, для приема полностью установленного соединения.

Параметр `sockd` является дескриптором созданного и настроенного сокета, предварительного переведенного в пассивный (слушающий) режим с помощью системного вызова `listen()`.

Системный вызов `accept` требует предварительной настройки адреса сокета с помощью системного вызова `bind()`.

Параметр `cliaddr` служит для получения адреса клиента, установившего логическое соединение, и должен содержать указатель на структуру, в которую будет занесен этот адрес.

Параметр `clilen` содержит указатель на целую переменную, которая после возвращения из вызова будет содержать фактическую длину адреса клиента. Заметим, что перед вызовом эта переменная должна содержать максимально допустимое значение такой длины. Если параметр `cliaddr` имеет значение `NULL`, то и параметр `clilen` может иметь значение `NULL`.

Возвращаемое значение

Системный вызов возвращает при нормальном завершении дескриптор присоединенного сокета, созданного при установлении

соединения для последующего общения клиента и сервера, и значение -1 при возникновении ошибки.

Программа 10-04, реализующая простой TCP-сервер для сервиса echo

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
void main()
{
    int sockfd, newsockfd; /* Дескрипторы для слушающего и
присоединенного сокетов */
    int clien; /* Длина адреса клиента */
    int n; /* Количество принятых символов */
    char line[1000]; /* Буфер для приема информации */
    struct sockaddr_in servaddr, cliaddr; /* Структуры для размещения
полных адресов сервера и клиента */
    /* Создаем TCP-сокет */
    if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
        perror(NULL);
        exit(1);
    }
    /* Заполнение структуры для адреса сервера: семейство
протоколов TCP/IP, сетевой интерфейс – любой, номер порта 51000.
Поскольку в структуре содержится дополнительное ненужное поле,
которое должно быть нулевым, обнуляем ее всю перед заполнением */
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family= AF_INET;
    servaddr.sin_port= htons(51000);
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    /* Настраиваем адрес сокета */
    if(bind(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0){
```

```

    perror(NULL);
    close(sockfd);
    exit(1);
}
/* Переводим созданный сокет в пассивное (слушающее) состояние.
Глубину очереди для установленных соединений описываем значением
5 */
if(listen(sockfd, 5) < 0){
    perror(NULL);
    close(sockfd);
    exit(1);
}
/* Основной цикл сервера */
while(1){
    /* В переменную cliilen заносим максимальную длину ожидаемого
адреса клиента */
    cliilen = sizeof(cliaddr);
    /* Ожидание полностью установленного соединения на
слушающем сокете. При нормальном завершении в структуре cliaddr
будет лежать полный адрес клиента, установившего соединение, а в
переменной cliilen – его фактическая длина. Вызов вернет дескриптор
присоединенного сокета, через который будет происходить общение
с клиентом. Информация о клиенте у нас в дальнейшем никак не
используется, поэтому вместо второго и третьего параметров
можно было поставить значения NULL. */
    if((newsockfd = accept(sockfd, (struct sockaddr *) &cliaddr, &cliilen))
< 0){
        perror(NULL);
        close(sockfd);
        exit(1);
    }
    /* В цикле принимаем информацию от клиента до тех пор, пока
не произойдет ошибка (вызов read() вернет отрицательное значение)
или клиент не закроет соединение (вызов read() вернет значение 0).
Максимальную длину одной порции данных от клиента ограничим 999
символами. В операциях чтения и записи пользуемся дескриптором
присоединенного сокета, т. е. значением, которое вернул вызов
accept().*/

```

```

while((n = read(newsockfd, line, 999)) > 0){
    /* Принятые данные отправить обратно */
    if((n = write(newsockfd, line, strlen(line)+1)) < 0){
        perror(NULL);
        close(sockfd);
        close(newsockfd);
        exit(1);
    }
}
/* Если при чтении возникла ошибка – завершение работы */
if(n < 0){
    perror(NULL);
    close(sockfd);
    close(newsockfd);
    exit(1);
}
/* Закрытие дескриптора присоединенного сокета и ожидание
нового соединения */
close(newsockfd);
}
}

```

Листинг 10-04. Программа 10-04. Пример простого TCP-сервера для сервиса echo

10.20 Применение интерфейса сетевых вызовов для других семейств протоколов. UNIX Domain протоколы.

Файлы типа «сокет»

Рассмотренный выше интерфейс умеет работать не только со стеком протоколов TCP/IP, но и с другими семействами протоколов. При этом требуется лишь незначительное изменение написанных с его помощью программ. Рассмотрим действия, которые необходимо выполнить для модернизации написанных для TCP/IP программ под другое семейство протоколов.

Изменяется тип сокета, поэтому для его точной спецификации нужно задавать другие параметры в системном вызове socket().

В различных семействах протоколов применяются различные адресные пространства для удаленных и локальных адресов сокетов. Поэтому меняется состав структуры для хранения полного адреса сокета, название ее типа, наименования полей и способ их заполнения.

Описание типов данных и предопределенных констант будет находиться в других include-файлах, поэтому потребуется заменить include-файлы `<netinet/in.h>` и `<arpa/inet.h>` на файлы, относящиеся к выбранному семейству протоколов.

Может измениться способ вычисления фактической длины полного адреса сокета и указания его максимального размера.



Рис. 10.9. Схема работы TCP-сервера с параллельной обработкой запросов

Семейство UNIX Domain протоколов предназначено для общения локальных процессов с использованием интерфейса системных вызовов. Оно содержит один потоковый и один датаграммный протокол. Никакой сетевой интерфейс при этом не используется, а вся передача информации реально происходит через адресное пространство ядра операционной системы. Многие программы, взаимодействующие и с локальными, и с удаленными процессами (например, X-Windows), для локального общения используют этот стек протоколов.

Поскольку общение происходит в рамках одной вычислительной системы, в полном адресе сокета его удаленная часть отсутствует. В качестве адресного пространства портов – локальной части адреса – выбрано адресное пространство, совпадающее с множеством всех допустимых имен файлов в файловой системе. При этом в качестве имени сокета требуется задавать имя несуществующего еще файла в директории, к которой у вас есть права доступа как на запись, так и на чтение. При настройке адреса (системный вызов **bind()**) под этим именем будет создан файл типа «сокет» – последний еще неизвестный нам тип файла. Этот файл для сокетов играет роль файла-метки типа FIFO для именованных **pip**'ов.

Если на вашей машине функционирует **X-Windows**, то вы сможете обнаружить такой файл в директории с именем **/tmp/.X11-UNIX** – это файл типа «сокет», служащий для взаимодействия локальных процессов с оконным сервером.

Для хранения полного адреса сокета используется структура следующего вида, описанного в файле **<sys/un.h>**:

```
struct sockaddr_un{
    short sun_family;
    /* Избранное семейство протоколов – всегда AF_UNIX */

    char sun_path[108];
    /* Имя файла типа "сокет" */
};
```

Выбранное имя будет копироваться внутрь структуры, используя функцию **strcpy()**.

Фактическая длина полного адреса сокета, хранящегося в структуре с именем `my_addr`, может быть вычислена следующим образом: **`sizeof(short)+strlen(my_addr.sun_path)`**. В Linux для этих целей можно использовать специальный макрос языка Си.

SUN_LEN(struct sockaddr_un*)

Ниже приведены тексты переписанных под семейство UNIX Domain протоколов клиента и сервера для сервиса `echo` (программы 11-05 и 11-06), общающиеся через датаграммы. Клиент использует сокет с именем `AAAA` в текущей директории, а сервер – сокет с именем `BBBB`. Как следует из описания типа данных, эти имена (полные или относительные) не должны по длине превышать 107 символов. Комментарии даны лишь для изменений по сравнению с программами 10-01 и 10-02.

Программа 10-05. UNIX Domain протокол сервера для сервиса `echo`, общающегося через датаграммы

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h> /* Новый include-файл вместо netinet/in.h и
arpa/inet.h */
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
int main()
{
    int sockfd;
    int clilen, n;
    char line[1000];
    struct sockaddr_un servaddr, cliaddr; /* новый тип данных под
адреса сокетов */
    if((sockfd = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0) /* Изменен тип
семейства протоколов */
    {
        perror(NULL);
        exit(1);
    }
```



```

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sun_family = AF_UNIX; /* Изменен тип семейства
    протоколов и имя поля в структуре */
    strcpy(servaddr.sun_path, "BBBB"); /* Локальный адрес сокета
    сервера – BBBB – в текущей директории */
    if(bind(sockfd, (struct sockaddr *) &servaddr, SUN_LEN(&servaddr)) <
    0) /* Изменено вычисление фактической длины адреса */
    {
        perror(NULL);
        close(sockfd);
        exit(1);
    }
    while(1) {

        clilen = sizeof(struct sockaddr_un); /* Изменено вычисление
        максимальной длины для адреса клиента */
        if((n = recvfrom(sockfd, line, 999, 0, (struct sockaddr *) &cliaddr,
        &clilen)) < 0)
        {

            perror(NULL);
            close(sockfd);
            exit(1);

        }
        if(sendto(sockfd, line, strlen(line), 0, (struct sockaddr *) &cliaddr,
        clilen) < 0)
        {
            perror(NULL);
            close(sockfd);
            exit(1);
        }
    }
    return 0;
}

```

Листинг 10-05. UNIX Domain протокол сервера для сервиса echo, общающегося через датаграммы

Программа 10-06. UNIX Domain протокол клиента для сервиса echo, общающегося через датаграммы

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h> /* Новый include-файл вместо netinet/in.h и
arpa/inet.h */
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <iostream>
using namespace std;
int main() /* Аргументы командной строки не нужны, так как сервис
является локальным, и не нужно указывать, к какой машине мы
обращаемся с запросом */
{
    int sockfd;
    int n, len;
    char sendline[1000], recvline[1000];
    struct sockaddr_un servaddr, cliaddr; /* новый тип данных под адреса
сокетов */
    if((sockfd = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0)
        /* Изменен тип семейства протоколов */
    {
        perror(NULL);
        exit(1);
    }
    bzero(&cliaddr, sizeof(cliaddr));
    cliaddr.sun_family= AF_UNIX; /* Изменен тип семейства
протоколов и имя поля в структуре */
    strcpy(cliaddr.sun_path, "AAAA"); /* Локальный адрес сокета
клиента – AAAA – в текущей директории */
    if(bind(sockfd, (struct sockaddr *) &cliaddr, SUN_LEN(&cliaddr)) < 0)
        /* Изменено вычисление фактической длины адреса */
    {
        perror(NULL);
        close(sockfd);
    }
}
```

```

    exit(1);
}
bzero(&servaddr, sizeof(servaddr));
servaddr.sun_family = AF_UNIX; /* Изменен тип семейства
протоколов и имя поля в структуре */
strcpy(servaddr.sun_path, "BBBB"); /* Локальный адрес сокета
сервера – BBBB – в текущей директории */
cout<<"String => "<<endl;
fgets(sendline, 1000, stdin);
if(sendto(sockfd, sendline, strlen(sendline)+1, 0, (struct sockaddr *)
&servaddr, SUN_LEN(&servaddr)) < 0) /* Изменено вычисление
фактической длины адреса */
{
    perror(NULL);
    close(sockfd);
    exit(1);
}
if((n = recvfrom(sockfd, recvline, 1000, 0,
(struct sockaddr *) NULL, NULL)) < 0){
    perror(NULL);
    close(sockfd);
    exit(1);
}
recvline[n] = 0;
cout<< recvline<< endl;
close(sockfd);
return 0;
}

```

Листинг 10-06. Программа 10-06. UNIX Domain протокол клиента для сервиса echo, общающегося через датаграммы

Порядок выполнения лабораторной работы:

1. Наберите и откомпилируйте программу. Перед запуском «узнайте у своего системного администратора», запущен ли в системе стандартный UDP-сервис echo, и если нет, попросите стартовать его. Запустите программу с запросом к сервису своего компьютера, к сервисам других компьютеров. Если в качестве IP-

адреса указать несуществующий адрес, адрес выключенной машины или машины, на которой не работает сервис echo, то программа бесконечно блокируется в вызове `recvfrom()`, ожидая ответа. Протокол UDP не является надежным протоколом. Если датаграмму доставить по назначению не удалось, то отправитель никогда об этом не узнает!

2. Наберите и откомпилируйте программу. Запустите ее на выполнение.

3. Модифицируйте текст программы UDP-клиента (программа 10-1), заменив номер порта с 7 на 51000. Запустите клиента с другого виртуального терминала или с другого компьютера и убедитесь, что клиент и сервер взаимодействуют корректно.

4. Наберите и откомпилируйте программу 10-3. Перед запуском «узнайте у своего системного администратора», запущен ли в системе стандартный TCP-сервис echo, и, если нет, попросите это сделать. Запустите программу с запросом к сервису своего компьютера, к сервисам других компьютеров. Если в качестве IP-адреса указать несуществующий адрес или адрес выключенной машины, то программа сообщит об ошибке при работе вызова `connect()` (правда, возможно, придется подождать окончания `timeout`'а). При задании адреса компьютера, на котором не работает сервис echo, об ошибке станет известно сразу же. Протокол TCP является надежным протоколом. Если логическое соединение установить не удалось, то отправитель будет знать об этом.

5. Наберите и откомпилируйте программу 10-4. Запустите ее на выполнение.

6. Модифицируйте текст программы TCP-клиента (программа 10-3.), заменив номер порта с 7 на 51000. Запустите клиента с другого виртуального терминала или с другого компьютера и убедитесь, что клиент и сервер взаимодействуют корректно.

7. В программах 10-3 и 10-4 сервер осуществлял последовательную обработку запросов от разных клиентов. При таком подходе клиенты могут подолгу простаивать после установления соединения, ожидая обслуживания. Поэтому обычно применяется схема псевдопараллельной обработки запросов. После приема установленного соединения сервер порождает процесс-ребенок, которому и поручает дальнейшую работу с клиентом. Процесс-родитель закрывает присоединенный сокет и уходит на

ожидание нового соединения. Схематично организация такого сервера изображена на рис. 10.9. Напишите, откомпилируйте и запустите такой параллельный сервер. Убедитесь в его работоспособности. Не забудьте о необходимости удаления зомби-процессов.

8. Наберите программы 10-5 и 10-6, откомпилируйте их и убедитесь в работоспособности.

9. По аналогии с программами 10-5 и 10-6 модифицируйте тексты программ TCP клиента и сервера для сервиса echo (программа 10-3 и программа 10-4) для потокового общения в семействе UNIX Domain протоколов. Откомпилируйте их и убедитесь в правильном функционировании.

Контрольные вопросы

1. Краткая история семейства протоколов TCP/IP.
2. Общие сведения об архитектуре семейства протоколов TCP/IP.
3. Уровень сетевого интерфейса. Уровень Internet.
4. Протоколы IP, ICMP, ARP, RARP.
5. Internet-адрес. Транспортный уровень.
6. Протоколы TCP и UDP. TCP и UDP сокеты.
7. Адресные пространства портов.
8. Понятие encapsulation.
9. Использование модели клиент-сервер для взаимодействия удаленных процессов.
10. Организация связи между удаленными процессами с помощью датаграмм.
11. Сетевой порядок байт. Функции htons(), htonl(), ntohs(), ntohl().
12. Функции преобразования IP-адресов inet_ntoa(), inet_aton().
13. Функция bzero().
14. Создание сокета. Системный вызов socket().
15. Адреса сокетов. Настройка адреса сокета.
16. Системный вызов bind().
17. Системные вызовы sendto() и recvfrom().
18. Определение IP-адресов для вычислительного комплекса.
19. Организация связи между процессами с помощью установки логического соединения.

20. Установление логического соединения. Системный вызов `connect()`.
21. Как происходит установление виртуального соединения.
22. Системный вызов `listen()`.
23. Системный вызов `accept()`.
24. Пример простого TCP-сервера.
25. Применение интерфейса сетевых вызовов для других семейств протоколов. UNIX.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Назаров С.В. Операционные системы: учебное пособие для студентов вузов, обучающихся по направлению подготовки 080700 «Бизнес-информатика» / С.В. Назаров, Л.П. Гудыно, А.А. Кириченко; - М.: Кнорус, 2012. -371 с.
2. Назаров С.В. Современные операционные системы: учебное пособие / С.В. Назаров, А.И. Широков - 2-е изд., испр. и доп. — М.: БИНОМ. Лаборатория знаний, 2013. - 367 с.

Дополнительная литература

1. Таненбаум Э. Современные операционные системы/ Э. Таненбаум, Х. Босх - 4-е изд. – СПб.: Питер, 2016. – 1120 с.
2. Таненбаум Э. Архитектура компьютера/ Э. Таненбаум - 5-е изд. – СПб.: Питер, 2013. – 844 с.
3. Таненбаум Э. Операционные системы. Разработка и реализация. Классика CS/ Э. Таненбаум, А. Вудхалл - 3-е изд. – СПб.: Питер, 2007. – 704 с.
4. Олифер В.Г. Компьютерные сети. Принципы, технологии, протоколы: учебник для вузов/В.Г. Олифер - 3-е изд. - СПб.: Питер, 2010. – 958 с.

Ссылки на Internet ресурсы

1. <http://www.intuit.ru/studies/courses/2249/52/info> – НОУ ИНТУИТ Академия Intel: Основы операционных систем. Практикум
2. www.linuxcenter.ru – магазин, дистрибутивы Linux, подборка статей по Linux, новости в мире Linux;
3. www.linux-online.ru – интернет магазин, дистрибутивы Linux, книги по администрированию, ссылки;
4. www.citfonim.ru – сервер информационных технологий, содержит большой объем информации по информационным технологиям, в том числе и по Linux;
5. www.linux.webclub.ru – новости в мире Linux;

6. www.linuxrsp.ru – Интернет проект “Все о Linux по русски” (статьи, рассылки, документация и т.п.);
7. Операционная система Linux: Часть 2. Терминал и командная строка;
8. Учебное пособие от компании IBM <http://www.ibm.com/developerworks/ru/edu/l-dw-linuxredbook2.html>;
9. Программирование на Shell (UNIX) // Библиотека <http://www.linuxcenter.ru/lib/books/shell/>;
10. Novell Open Suse – официальный сайт проекта на русском <https://ru.opensuse.org/>