РОСЖЕЛДОР

Федеральное государственное бюджетное образовательное учреждение высшего образования
«Ростовский государственный университет путей сообщения»

(ФГБОУ ВО РГУПС)

Н.А. Москат

ОПЕРАЦИОННЫЕ СИСТЕМЫ И КРОССПЛАТФОРМЕННОЕ ПРОГРАММИРОВАНИЕ. ОПЕРАЦИОННАЯ СИСТЕМА LINUX

Учебно-методическое пособие
для лабораторных и практических работ

Ростов-на-Дону
2017

УДК 681.3.06(07) + 06

Рецензент – доктор технических наук, профессор М.А. Бутакова

Москат, Н.А.

Операционные системы и кроссплатформенное программирование. Операционная система LINUX: учебно-методическое пособие для лабораторных и практических работ / Н.А. Москат; ФГБОУ ВО РГУПС. – Ростов н/Д, 2017. – 98 с.

Представлены лабораторные работы по работе с операционной системой Linux. Приведен теоретический материал по различным классам задач операционной системы, дано обширное количество примеров. К каждой работе имеется задание и контрольные вопросы.

Предназначено для студентов и магистрантов направлений «Информатика и вычислительная техника», «Информационные системы и технологии» и «Мехатроника и робототехника», а также для студентов, аспирантов и магистрантов всех специальностей, изучающих дисциплины «Операционные системы», «Операционные системы и кроссплатформенное программирование», «Операционные системы реального времени» «Операционная система LINUX», «Инструментальные средства информационных систем (Операционные системы)» и смежные дисциплины. Изложенный материал будет полезен при курсовом и дипломном проектировании, а также аспирантам, магистрам и соискателям.

Одобрено к изданию кафедрой «Вычислительная техника и автоматизированные системы управления».

ОГЛАВЛЕНИЕ

ЛАБОРАТОРНАЯ РАБОТА № 1. ОСНОВНЫЕ ПРИНЦИПЫ ФУНКЦИОНИРОВАНИЯ ОПЕРАЦИОННОЙ СИСТЕМЫ LINUX

1. Цель работы

Целью работы является изучение архитектуры и принципов функционирования многопользовательской многозадачной операционной системы Linux.

2. Теоретическая часть

Система включает следующие основные компоненты.

Ядро. Выполняет функции управления памятью, процессорами. Осуществляет диспетчеризацию выполнения всех программ и обслуживание внешних устройств. Все действия, связанные с вводом/выводом и выполнением системных операций, выполняются с помощью системных вызовов. Системные вызовы реализуют программный интерфейс между программами и ядром. Имеется возможность динамического конфигурирования ядра.

Диспетчер процессов Init. Активизирует процессы, необходимые для нормальной работы системы и производит их начальную инициализацию. Обеспечивает завершение работы системы, организует сеансы работы пользователей, в том числе, для удаленных терминалов.

Интерпретатор команд Shell. Анализирует команды, вводимые с терминала либо из командного файла, и передает их для выполнения в ядро системы. Команды обычно имеют аргументы и параметры, которые обеспечивают модернизацию выполняемых действий. Shell является также языком программирования, на котором можно создавать командные файлы (shell-файлы). При входе в ОС пользователь получает копию интерпретатора shell в качестве родительского процесса. Далее, после ввода команды пользователем создается порожденный процесс, называемый процессом-потомком. Т.е. после запуска ОС каждый новый процесс функционирует только как процесс – потомок уже существующего процесса. В ОС Linux имеется возможность динамического порождения и управления процессами.

Shell – интерпретатор в соответствии с требованиями стандарта POSIX поддерживает графический экранный интерфейс, реализованный средствами языка программирования Tcl/Tk.

Обязательным в системе является интерпретатор Bash, полностью соответствующий стандарту POSIX. В качестве Shell может быть использована оболочка **mc** с интерфейсом, подобным NortonCommander.

Сетевой графический интерфейс X-сервер (X-Windows). Обеспечивает поддержку графических оболочек.

Графические оболочки KDE, Gnome. Отличительными свойствами KDE являются: минимальные требования к аппаратуре, высокая надежность, интернационализация. Базовые библиотеки KDE (qt, kde-libs) признаны одними из лучших продуктов по созданию графического интерфейса, обеспечивают простое написание программ с использованием передовых технологий. Gnome имеет развитые графические возможности, но более требователен к аппаратным средствам.

Сетевая поддержка NFS, SMB, TCP/IP. NFS – программный комплекс PC-NFS (NetworkFileSystem) для выполнения сетевых функций. PC-NFS ориентирован для конкретной ОС персонального компьютера (PC) и включает драйверы для работы в сети и дополнительные утилиты. SMB – сетевая файловая система, совместимая с Windows NT. TCP/IP – протокол контроля передачи данных (TransferControlProtocol/InternetProtocol). Сеть по протоколам TCP/IP является неотъемлемой частью ОС семейства UNIX. Поддерживаются любые сети, от локальных до Internet, с использованием только встроенных сетевых средств.

Инструментальные средства программирования. Основой средств программирования является компилятор GCC или его экспериментальные версии EGCS и PGCC для языков C и C++; модули поддержки других языков программирования (Obective C, Фортран, Паскаль, Modula-3, Ада, Java и др.); интегрированные среды и средства визуального проектирования: Kdevelop, Xwpe; средства адаптации привязки программ AUTOCONFIG, AUTOMAKE.

2.1 Регистрация пользователя в системе

Для входа пользователя с терминала в многопользовательскую операционную систему LINUX необходимо зарегистрироваться в качестве пользователя. Для этого нужно после сообщения **Login:**

ввести системное имя пользователя, например, «student". Если имя задано верно, выводится запрос на ввод пароля: **Password:**

Наберите пароль "student" и нажмите клавишу *Enter*.

Если имя или пароль указаны неверно, сообщение *login* повторяется. Значение пароля проверяется в системном файле *password*, где приводятся и другие сведения о пользователях. После правильного ответа появляется приветствие LINUX и приглашение:

student@linux:>

Вы получили доступ к ресурсам ОС LINUX.

2.2 Выход из системы

exit – окончание сеанса пользователя.

2.3 Выполнение простых команд

Формат команд в ОС LINUX следующий:

имя команды [аргументы] [параметры] [метасимволы]

Имя команды может содержать любое допустимое имя файла; аргументы – одна или несколько букв со знаком минус (-); параметры – передаваемые значения для обработки; метасимволы интерпретируются как специальные операции. В квадратных скобках указываются необязательные части команд.

Введите команду **echo**, которая выдает на экран свои аргументы:

echo good morning

инажмитеклавишу*Enter*. На экране появится приветствие "*goodmorning*" – аргумент команды **echo.** Командный интерпретатор *shell* вызвал команду **echo,** реализованную в виде программы на языке СИ, и передал ей аргументы. После этого интерпретатор команд вывел знак-приглашение. Синтаксис команды **echo**:

echo [-n] [arg1] [arg2] [arg3]...

Команда помещает в стандартный вывод свои аргументы, разделенные пробелами и завершаемые символом перевода строки. При наличии флага *-n* символ перевода строки исключается.

who [am i] – получение информации о работающих пользователях.

В квадратных скобках указываются аргументы команды, которые можно опустить. Ответ представляется в виде таблицы, которая содержит следующую информацию:

- идентификатор пользователя;

- идентификатор терминала;

- дата подключения;

- время подключения.

date – вывод на экран текущей даты и текущего времени.

cal [[месяц]год] – календарь; если календарь не помещается на одном экране, то используется команда **cal год | more** и клавишей пробела производится постраничный вывод информации.

man<название команды> – вызов электронного справочника об указанной команде. Выход из справочника – нажатие клавиши Q.

Команда **manman** сообщает информацию о том, как пользоваться справочником.

tty – сообщение имени специального файла стандартного вывода, соответствующего терминалу пользователя.

cat<имя файла> – вывод содержимого файла на экран. Команда **cat> text.1** создает новый файл с именем text.1, который можно заполнить символьными строками, вводя их с клавиатуры. Нажатие клавиши *Enter* создает новую строку. Завершение ввода – нажатие *Ctrl – d*. Команда **cat text.1 > text.2** пересылает содержимое файла text.1 в файл text.2. Слияние файлов осуществляется командой **cat text.1 text.2 > text.3**.

ls [-alrstu] [имя] – вывод содержимого каталога на экран. Если аргумент не указан, выдается содержимое текущего каталога.

Аргументы команды:

-a – выводит список всех файлов и каталогов, в том числе и скрытых;

-l – выводит список файлов в расширенном формате, показывая тип каждого элемента, полномочия, владельца, размер и дату последней модификации;

- r – выводит список в порядке, обратном заданному;

- s – выводит размеры каждого файла;

- t – перечисляет файлы и каталоги в соответствии с датой их последней модификации;

- u – перечисляет файлы и каталоги в порядке, обратном их последней модификации.

rm<имя файла> – удаление файла (файлов). Команда**rm text.1 text.2 text.3**удаляетфайлы text.1, text.2, text.3. Другие варианты этой команды – **rmtext.[123]** или **rmtext.[1-3].**

wc [имя файла] – вывод числа строк, слов и символов в файле.

clear – очистка экрана.

2.4 Группирование команд

Группы команд или сложные команды могут формироваться с помощью специальных символов (метасимволов):

& – процесс выполняется в фоновом режиме, не дожидаясь окончания предыдущих процессов;

? – шаблон, распространяется только на один символ;

* – шаблон, распространяется на все оставшиеся символы;

| – программный канал – стандартный вывод одного процесса является стандартным вводом другого;

> – переадресация вывода в файл;

< – переадресация ввода из файла;

; – если в списке команд команды отделяются друг от друга точкой с запятой, то они выполняются друг за другом;

&& – эта конструкция между командами означает, что последующая команда выполняется только при нормальном завершении предыдущей команды (код возврата 0);

|| – последующая команда выполняется только, если не выполнилась предыдущая команда (код возврата 1);

() – группирование команд в скобки;

{ } – группирование команд с объединенным выводом;

[] – указание диапазона или явное перечисление (без запятых);

>> – добавление содержимого файла в конец другого файла.

Примеры.

Самый простой способ создания файла в linux – это оператор перенаправления >

> файл

Еще один способ создания файла

cat > file.txt – после выполнения команды можете вводить любые символы, которые нужно записать в файл, для сохранения нажмите Ctrl+D.

Можно создать файл еще и так:

echo "Это строка" > файл.txt

Также можно не только перезаписывать файл, а дописывать в него данные, с помощью перенаправления оператора >>. Если файла не существует, будет создан новый, а если существует, то строка запишется в конец.

$ echo "Это текстовая строка" > файл.txt
$ echo "Это вторая текстовая строка" >> файл.txt

who | wc – подсчет количества работающих пользователей командой **wc** (wordcount – счет слов);

cat text.1 > text.2 – содержимое файла text.1 пересылается в файл text.2;

mailstudent< file.txt – электронная почта передает файл file.txt всем пользователям, перечисленным в командной строке;

cat text.1,text.2 – просматриваются файлы text.1 и text.2;

cat text.1 >> text.2 – добавление файла text.1 в конец файла text.2;

rm text.* – удаление всех файлов с именем text;

{cat text.1; cat text.2} | lpr – просмотр файлов text.1 и text.2 и вывод их на печать;

ps [al] [number] – команда для вывода информации о процессах:

-a – вывод информации обо всех активных процессах, запущенных с вашего терминала;

-l – полная информация о процессах;

number – номер процесса.

Команда **ps** без параметров выводит информацию только об активных процессах, запущенных с данного терминала, в том числе и фоновых. На экран выводится подробная информация обо всех активных процессах в следующей форме:

F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD

1 S 200 210 7 0 2 20 80 30 703a 03 0:07 cc

1 R 12 419 7 11 5 20 56 20 03 0:12 ps

F – флаг процесса (1 – в оперативной памяти, 2 – системный процесс, 4 – заблокирован в ОЗУ, 20 – находится под управлением другого процесса, 10 – подвергнут свопингу);

S – состояние процесса (O – выполняется процессором , S – задержан, R – готов к выполнению, I – создается);

UID – идентификатор пользователя;

PID – идентификатор процесса;

PPID – номер родительского процесса;

C – степень загруженности процессора;

PRI – приоритет процесса, вычисляется по значению переменной NICE и чем больше число, тем меньше его приоритет;

NI – значение переменной NICE для вычисления динамического приоритета, принимает величины от 0 до 39;

ADDR – адрес процесса в памяти;

SZ – объем ОЗУ, занимаемый процессом;

WCHAN – имя события, до которого процесс задержан, для активного процесса – пробел;

TTY – номер управляющего терминала для процесса;

TIME – время выполнения процесса;

CMD – команда, которая породила процесс.

nice [-приращение приоритета] команда[аргументы] – команда изменения приоритета. Каждое запущенное задание (процесс) имеет номер приоритета в диапазоне от 0 до 39, на основе которого ядро вычисляет фактический приоритет, используемый для планирования процесса. Значение 0 представляет наивысший приоритет, а 39 – самый низший. Увеличение номера приоритета приводит к понижению приоритета, присвоенного процессу. Команда **nice -10 ls -l** увеличивает номер приоритета, присвоенный процессу **ls -l** на 10.

renice 5 1836 – команда устанавливает значение номера приоритета процесса с идентификатором 1836 равным 5. Увеличить приоритет процесса может только администратор системы.

kill [-sig] <идентификатор процесса> – прекращение процесса до его программного завершения. sig – номер сигнала. Sig = -15 означает программное (нормальное) завершение процесса, номер сигнала = -9 – уничтожение процесса. По умолчанию sig= -9. Вывести себя из системы можно командой kill -9 0. Пользователь с низким приоритетом может прервать процессы, связанные только с его терминалом.

mc – вызов файлового менеджера (программы – оболочки) *MidnightCommander*, аналогичного *NortonCommander*.

sort [-dr] – сортировка входных файлов и вывод результата на экран.

3. Задание к лабораторной работе№1.

1. Ознакомиться с теоретической частью к лабораторной работе.
2. Зарегистрироваться в системе LINUX.
3. Просмотреть текущую дату.
4. Определить день недели, в который Вы родились.
5. Получить подробную информацию обо всех активных процессах.
6. Создать два текстовых файла (с расширением TXT) и командой CAT просмотреть их на экране.
7. Получить информацию о работающих пользователях, подсчитать их количество и запомнить в файле.
8. Объединить текстовые файлы в единый файл и посмотреть его на экране.
9. Посмотреть приоритет своего процесса и уменьшить скорость его выполнение за счет повышения номера приоритета.

4. Контрольные вопросы

1. Перечислите основные функции и назначение многопользовательской многозадачной операционной системы LINUX и ее отличительные особенности от однопрограммной системы DOS.

2. Какое назначение имеет ядро системы и интерпретатор команд?

3. В чем заключается понятие "процесс" и какие операции можно выполнить над процессами?

4. Как задаются и выполняются простые и сложные команды?

5. Какие функции выполняет командный интерпретатор *Shell*?

ЛАБОРАТОРНАЯ РАБОТА № 2. ПРОГРАММИРОВАНИЕ В LINUX. ТЕКСТОВЫЙ РЕДАКТОР VI.

1. Цель работы

Цель работы – научиться отлаживать, собирать, запускать программы (проекты) в Linux. Изучить основные возможности текстового редактора VI. Научиться работать с файлами (создавать, редактировать, удалять) с помощью указанного редактора.

2. Теоретическая часть

Создание любой программы начинается с постановки задачи, проектирования и написания исходного кода (source code). Обычно исходный код программы записывается в один или несколько файлов, которые называют *исходными файлами* или *исходниками*.

Исходные файлы обычно создаются и набираются в текстовом редакторе. В принципе, для написания исходных кодов подойдет любой текстовый редактор. Но желательно, чтобы это был редактор с "подсветкой" синтаксиса, т. е. выделяющий визуально ключевые слова используемого языка программирования. В результате исходный код становится более наглядным, а программист делает меньше опечаток и ошибок.

В современных дистрибутивах Linux представлен большой выбор текстовых редакторов. Наибольшей популярностью среди программистов пользуются редакторы двух семейств:

- *vi* (Visual Interface) – полноэкранный редактор, созданный Биллом Джоем (Bill Joy) в 1976 г. С тех пор было написано немало клонов vi. Практически все Unix-подобные системы комплектуются той или иной версией этого текстового редактора. Наиболее популярные клоны vi в Linux – vim (Vi IMproved), Elvis и nvi;
- *Emacs* (Editor MACroS) – текстовый редактор, разработанный Ричардом Столлманом (Richard Stallman). Из всех существующих версий Emacs наиболее популярными являются GNU Emacs и XEmacs.

Среди других распространенных в Linux редакторов следует отметить pico (PIne COmposer), jed и mcedit (Midnight Commander EDITor). Они не обладают мощью vi или Emacs, но достаточно просты и удобны в использовании. В Linux также имеется множество текстовых редакторов с графическим интерфейсом: kate, gedit, nedit, bluefish, jedit (этот список можно продолжать очень долго). Редакторы vim и GNU Emacs тоже имеют собственные графические расширения.

2.1 Работа в текстовом редактор VI

VI – текстовый редактор, входящий в поставку практически всех операционных систем семейства Unix. Запуск его осуществляется командой:

 # vi [filename]

Работа с редактором весьма специфична. Первое, чему необходимо научиться, это выходить из редактора.. В vi Вам не помогут ни <escape>, ни <Alt+X>, ни даже <Ctrl+C>. Так же он не будет отображать то, что Вы попытаетесь набрать .Поэтому сначала необходимо ознакомиться со следующими командами:

Команда	Описание
:q!	Выход без сохранения
:w	Сохранение изменений
:w <filename>	Сохранение как <filename>
:wq	Выход с сохранением
:q	Выход, если файл не изменялся
i	Переход в режим вставки символов в позиции курсора.
a	Переход в режим вставки символов в позиции после курсора.
o	Вставка строки после текущей
O	Вставка строки над текущей
x	Удаление символа в позиции курсора
dd	Удаление текущей строки
u	Отмена последнего действия
<escape>	Возврат в режим команд

Команды, начинающиеся с символа «:», будут отображаться в нижней строке. Остальные выполняются «молча». Редактор vi имеет два режима работы – режим команд и режим редактирования.

Запускается он в командном режиме, так что все нажатия на клавиши трактуются как команды. Нажатие клавиш "i", "a", "o", "O" и ряд других переводят vi в режим вставки, когда набираемые символы трактуются как текст и отображаются на экране.

Возврат к режиму команд выполняется клавишей <escape> или в некоторых случаях автоматически, например, при попытке передвинуть курсор левее первого символа в строке (в редакторе vim, являющемся модернизированным вариантом vi и частозаменяющем его в Linux, в этом случае редактор остается в режиме вставки). Автоматический переход в командный режим обычно сопровождается звуковым сигналом, как и ошибочная команда.

На рис.1 приведена схема , которая иллюстрирует взаимодействие этих режимов и способы перехода редактора между ними.

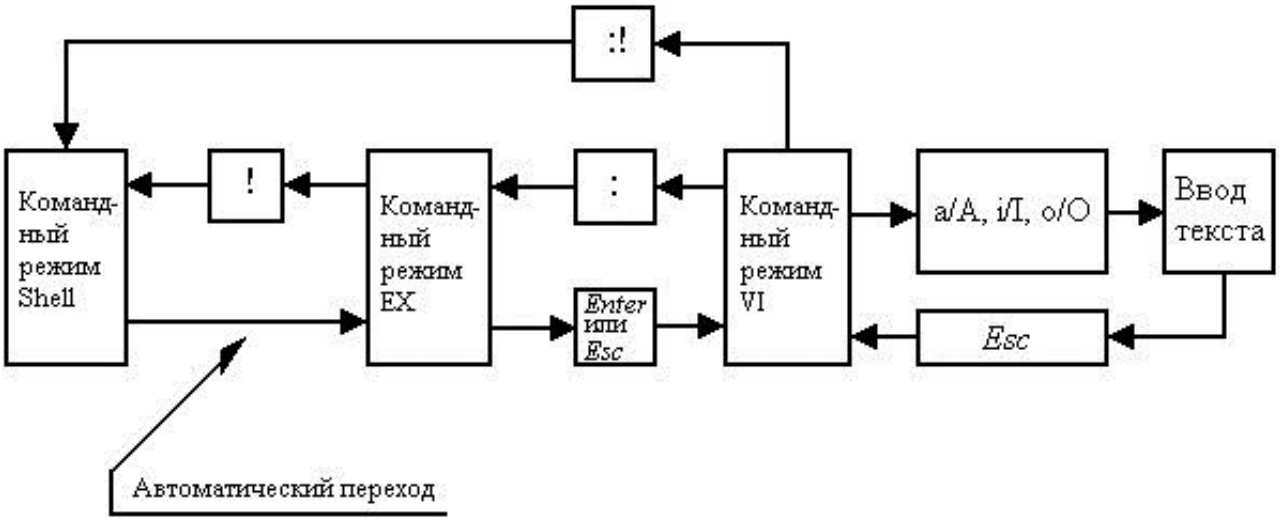

Рисунок 1 – Взаимодействие командных режимов редактора

В простейшем случае для вызова редактора нужно ввести команду **vi**text и нажать клавишу *Enter*. На экране появится:

```
$ vitext
_
~
.
.
"text"
```

Строка начинается знаком ~, знак _ определяет положение курсора. В данный момент пользователь находится в командном режиме **vi**. Перейти в режим ввода текста можно с помощью команд добавления текста, которые не отображаются на экране после их ввода:

a/A – ввод текста после курсора/после конца строки (append – присоединение);

i/I – вставка текста перед курсором/с 1-й позиции данной строки (insert – вставить);

o/O – образовать пустую строку ниже имеющейся / выше имеющейся.

Для выполнения команд (например, записи в файл, перемещения курсора) после введения текста или его части нужно перейти снова в командный режим **vi**, нажав клавишу *Esc*. После вызова **vi** нажмите клавишу a (ввод текста после курсора), не нажимая после этого клавишу *Enter*, и Вы попадете в режим ввода текста. Вводите текст, нажимая клавишу *Enter* в конце каждой строки (курсор в режиме ввода текста можно перемещать вправо, используя клавишу "*пробел*", и влево, используя клавишу *ВасkSpace*.

Переход в командный режим vi. Для перехода в командный режим vi нужно нажать клавишу *Esc*. Теперь редактор находится в командном режиме vi. В этом режиме выполняются следующие команды:

. – повторение последней команды;

u – аннулирование действия последней команды;

Переход в режим ex. Чтобы перейти к группе команд редактора ex (под именем ex редактор работает как строчно-ориентированный), нужно ввести символ : (двоеточие), команду и нажать <Enter> или Esc. Команды редактора ex начинаются с символа : и отображаются в нижней части экрана. После нажатия клавиши Esc или <Enter> происходит возврат (назад) в командный режим. Команды режима ex:

:w – запись текста в файл;

:r – чтение файла;

:e – редактирование нового файла;

:e! – выход без сохранения данного файла и редактирование нового;

:n – авторедактирование;

:wq – запись текста и выход из редактора;

:x – запись текста только при наличии в нем изменений;

:q! – оставить текст в рабочей области и закончить редактирование;

:ab – присвоение сокращений;

:map – определение ключей;

:set – изменение установочных режимов;

:s – выполнение замещений.

Переход в Shell. Редактор позволяет в процессе работы с ним выполнять команды ОС LINUX. Для этого нужно перейти в командный режим Shell с помощью команды !.

Рассмотрим пример. Определите текущее время командой date (вывод и установка даты) :! date. Здесь символ : означает переход в командный режим ex, а символ ! дает доступ к Shell. Для продолжительной работы с командами Shell можно вызвать командой :bash и после окончания работы вернуться в редактор vi, набрав CTRL-D.

Для возврата в командный режим vi нажмите клавишу *Enter*.

Команды, выполняемые в командном режиме VI

Изучим группу команд режима vi: перемещения курсора, добавления текста, поиска (частично), изменения и смещения текста, удаления, замены букв. Команды vi не отображаются на экране, кроме команд поиска, начинающихся со знаков / ? перемещение курсора, управление экраном дисплея, добавление текста.

Многие команды редактора выполняются только при определенном положении курсора, и нужно уметь пользоваться клавишами управления курсором (клавиши со стрелками <- , -> и т.д.). Кроме клавиши со стрелками для перемещения курсора можно использовать клавиши: CTRL-H – влево; CTRL-N – вниз; CTRL-P – вверх; SPACE – вправо.

Команды перемещения курсора:

h – на одну позицию влево;

l – на одну позицию вправо;

j – на одну позицию вниз;

k – на одну позицию вверх;

b – к первому символу предыдущего слова;

B – то же самое, что b, но игнорируются знаки пунктуации;

w – к первому символу следующего слова;

W – то же самое, что w, но игнорируются знаки пунктуации;

e – к последнему символу следующего слова;

E – то же самое, что e, но игнорируются знаки пунктуации;

(– к началу текущего предложения (предложение считается законченным, если после него есть два пробела или пустая строка);

) – к концу текущего предложения;

{ – к началу текущего раздела (разделителем раздела является пустая строка);

} – к концу текущего раздела;

[– к началу текущей секции;

] – к концу текущей секции;

^ – к первому отображаемому символу на текущей строке;

O – к началу текущей строки;

$ – к концу текущей строки;

H – к началу экрана;

M – на середину экрана;

L – к концу экрана;

nG – к строке с номером n (на последнюю строку, если номера n нет); % – к символу парной скобки, если курсор находится под одной из них.

Команды управления экраном:

^U – смещение текста на одну строку вверх (CTRL-U);

^D – смещение текста на одну строку вниз (CTRL-D);

^B – смещение текста на один кадр назад (CTRL-B);

^F – смещение текста на один кадр вперед (CTRL-F).

Чтобы переместить текущую строку:

- в верхнюю часть экрана нужно ввести команду z и нажать клавишу *Enter*;
- в середину экрана z;
- в нижнюю часть экрана z- .

Для очистки экрана от сообщений нужно использовать команды CTRL-R и CTRL-L; тексты в рабочей области при этом сохраняются.

Команды изменения текста:

cw – изменение слова;

cW – то же самое, что и cw, но игнорируются знаки пунктуации;

cO – от начала текущей строки;

c$ – до конца текущей строки;

cc – изменение всей строки;

c(– от начала текущего предложения;

c) – до конца текущего предложения;

c{ – от начала текущего раздела;

c} – до конца текущего раздела.

Для внесения изменений в текст необходимо: переместить курсор в нужную позицию; ввести команду изменения; без пробела набрать новый текст; нажать клавишу ESC.

Во всех командах можно использовать множители n, например для изменения пяти слов используется команда c5w.

Команды поиска начинаются косой чертой / (поиск вперед по тексту) или знаком ? (поиск назад); далее следует номер строки или ключевое слово. Команда заканчивается нажатием клавиши *Enter*.

Команды смещения текста:

<(или>(– к началу текущего предложения;

<)или>) – к концу текущего предложения;

<{или>{ – к началу текущего раздела;

<}или>} – к концу текущего раздела.

В командах смещения текста можно использовать множители, например может использоваться команда 2>> (сдвиг вправо). Смещение устанавливается командой: setsw=m. По умолчанию m=8. После того как курсор подведен к требуемой строке, нужно набрать символы << или >>.

Удаление, замена строчных букв на прописные и наоборот. Для удаления текста/фрагмента нужно переместить курсор в требуемую позицию и ввести команду удаления.

dw – до конца текущего слова;

dW – то же, что и dw, но игнорируются знаки пунктуации;

d^ – до 1-го видимого символа текущей строки;

dO – удаление начала строки;

d$ – удаление конца строки;

d(– до начала текущего предложения;

d) – до конца текущего предложения;

d{ – до начала текущего раздела;

d} – до конца текущего раздела;

dd – удаление всей строки;

dkw – удаление k слов;

dk)/dk} – удаление k предложений, k разделов;

kdd – удаление k строк.

Для удаления одиночного символа нужно подвести к нему курсор и набрать x (не d), а для удаления нескольких символов подряд набрать команду nx.

Для удаления текста от начала строки до определенного места и от определенного места до конца строки используются команды d^ и d$ соответственно.

Символ ~ используется для замены строчных букв на прописные и наоборот. Замена 1-й буквы в последней строке текста:

- Введите символ ((к началу текущего предложения).
- Наберите команду .~
- Восстановите текст командой u.

Определение текущей рабочей позиции в файле. После ввода пользователем в командном режиме CTRL-G в нижней части экрана появится статусная информация в соответствии с положением курсора в тексте, включающая: имя файла; сведения о проведенной ранее модификации; номер

текущей строки; общее число строк; расстояние курсора от начала файла (в процентах).

Для окончания работы с редактором введите в командном режиме :wq (запись текста из рабочей области в файл и окончание редактирования) и нажмите клавишу *Enter*. На экране появится сообщение о том, что Вы вышли из редактора и находитесь в Shell:

:wq<*Enter*>
/home/student>

Пример

Выполним небольшое практическое упражнение. Находясь в своем домашнем каталоге, запустите редактор командой:

 # vi test.txt

Далее нажмите "i", чтобы перейти в режим вставки. Теперь все нажатия на клавиши будут трактоваться как ввод текста, и символы будут отображаться на экране с позиции курсора. Наберите "Hello, world!". Постарайтесь не ошибаться, поскольку исправление ошибок, как и все остальное, имеет здесь свои особенности, о которых мы поговорим ниже. Нажмите <escape> для возврата в командный режим. Наберите ":wq" и нажмите <enter>. Убедитесь, что файл test.txt действительно создался. После этого небольшого задания Вам будет проще представлять то, о чем пойдет речь далее.

Аналогично команде "i", в режим вставки можно перейти, нажав клавишу"a". Единственное отличие – текст будет вставляться не перед символом, на котором находится курсор, а после него. Кроме того, режим вставки может быть вызван командами "o" и "O". Первая из них добавляет пустую строку после, а вторая – перед текущей строкой, и дальнейший ввод символов трактуется как ввод текста.

Чтобы удалить символ, нужно перейти в режим команд и над удаляемым символом нажать клавишу "x". В режиме вставки удалить только что введенный ошибочно символ можно клавишей , однако в vi таким образом может быть удалена только последняя непрерывно введенная последовательность символов. То есть если Вы откроете для редактирования наш тестовый файл со строкой "Hello, world", и добавите между словами слово "my": "Hello, myworld", то используя клавишу<backspace>, Вы сможете удалить только что введенные символы " my", а вот запятую и последующие символы удалить таким образом уже не удастся. В этом случае придется использовать команду "x".

Удалить целиком строку, на которой находится курсор, можно командой "dd" (просто нажмите два раза клавишу). Помните, что в vi строкой считается не экранная строка, а последовательность символов до перевода строки (\n). Если строка больше 80 символов (значение по умолчанию),т о она переносится на новую линию (строку экрана). Используя "dd", Вы удалит всю строку вне зависимости от того, на

скольких экранных линиях она размещается.

Чтобы определить, где находится конец строки, нажмите клавишу "$" (клавиши <Home>, <End> и т.п. тут не работают). При навигации по экрану (можно пользоваться стрелками, хотя есть и более "правильный" способ) курсор перемещается не по экранным линиям, а по строкам текста.

Если Вы что-то сделали не так, то отменить последнюю операцию можно командой "u". Эта команда отменяет только последнее действие, то есть ее повторное применение отменит только что сделанную отмену.

Учтите, что по команде "u" отменяется вся команда целиком.

Если ошибок получилось слишком много, можно выйти из редактора без сохранения сделанных изменений (команда ":q!").

2.2 Программирование в Linux

Проще всего изучать азы программирования на конкретном примере. Для начала создайте в своем текстовом редакторе файл myclock.c

```
#include <stdio.h>
#include <time.h>
int main (void)
{
time_t nt = time (NULL);
printf ("%s", ctime (&nt));
return 0;
}
```

Листинг 1.1. Программа myclock.c.

Это исходный код нашей первой программы. Рассмотрим его по порядку:

1.Заголовочный файл stdio.h делает доступными механизмы ввода-вывода стандартной библиотеки языка C. Нам он нужен для вызова функции printf().

2.Заголовочный файл time.h включается в программу, чтобы сделать доступными функции time() и ctime(), работающие с датой/временем.

3.Собственно программа начинается с функции main(), в теле которой создается переменная nt, имеющая тип time_t. Переменные этого типа предназначены для хранения числа секунд, прошедших с начала *эпохи отсчета компьютерного времени* (полночь 1 января 1970 г.).

4.Функция time() заносит в переменную nt текущее время.

5.Функция ctime() преобразовывает время, исчисляемое в секундах от начала *эпохи* (Epoch), в строку, содержащую привычную для нас запись даты и времени.

6. Полученная строка выводится на экран функцией printf().

7. Инструкция return 0; осуществляет выход из программы.

Компиляция

Чтобы запустить программу, ее необходимо сначала перевести с понятного человеку исходного кода в понятный компьютеру исполняемый код. Такой перевод называется *компиляцией* (compilation). Чтобы откомпилировать программу, написанную на языке C, нужно "пропустить" ее исходный код через

компилятор. В результате получается *исполняемый* (бинарный) код. Файл, содержащий исполняемый код, обычно называют *исполняемым файлом* или *бинарником* (binary).

Компилятором языка C в Linux обычно служит программа gcc (GNU C Compiler) из пакета компиляторов GCC (GNU Compiler Collection). Чтобы откомпилировать нашу программу, следует вызвать gcc, указав в качестве аргумента имя исходного файла:

$ gcc myclock.c

Если компилятор не нашел ошибок в исходном коде, то в текущем каталоге появится файл a.out. Теперь, чтобы выполнить программу, требуется указать командной оболочке путь к исполняемому файлу. Поскольку текущий каталог обычно обозначается точкой, то запуск программы можно осуществить следующим образом:

$ ./a.out

Wed Nov 8 03:09:01 2006

Исполняемые файлы программ обычно располагаются в каталогах, имена которых перечислены через двоеточие в особой переменной PATH. Чтобы просмотреть содержимое этой переменной, введите следующую команду:

$ echo $PATH

/usr/local/bin:/usr/bin:/bin:/usr/games:/usr/lib/qt4/bin

Если бинарник находится в одном из этих каталогов, то для запуска программы достаточно ввести ее имя (например, ls). В противном случае потребуется указание пути к исполняемому файлу.

Имя a.out не всегда подходит для программы. Один из способов исправить положе-ние – просто переименовать полученный файл:

$ mv a.out myclock

Но есть способ лучше. Можно запустить компилятор с опцией -o, которая позволяет явно указать имя файла на выходе:

$ gcc -o myclock myclock.c

Наша программа не содержит синтаксических ошибок, поэтому компилятор молча "проглатывает" исходный код. Проведем эксперимент, нарочно испортив программу. Для этого уберем в исходном файле первую инструкцию функции main(), которая объявляет переменную nt. Теперь снова попробуем откомпилировать полученный исходный код:

$ gcc -o myclock myclock.c

myclock.c: In function 'main':
myclock.c:6: error: 'nt' undeclared (first use in this function)
myclock.c:6: error: (Each undeclared identifier is reported only once
myclock.c:6: error: for each function it appears in.)

Комментарии используются программистами не только для пояснений и заметок. Перед компиляцией комментарии исключаются из исходного кода, как если бы их вообще не было. Поэтому программу myclock можно "испортить", просто закомментировав первую инструкцию функции main().

Как и ожидалось, компиляция не удалась. Обратите внимание, что находящийся в текущем каталоге исполняемый файл myclock – это "детище" предыдущей компиляции. Новый файл не был создан.
Очень важно научиться понимать сообщения об ошибках, выводимых компилятором. В нашем случае сообщается, что произошло "нечто" в файле myclock.c внутри функции main(). Далее говорится, что строка номер 6 содержит ошибку (errorerror): переменная nt не была объявлена к моменту ее первого использования в данной функции. В последних двух строках приводится пояснение: для каждой функции, где встречается необъявленный идентификатор (имя), сообщение об ошибке выводится только один раз.
Иногда вместо ошибки (error) выдается предупреждение (warning). В этом случае компиляция не останавливается, но до сведения программиста доводится информация о потенциально опасной конструкции исходного кода.
Теперь верните недостающую строку обратно или раскомментируйте, поскольку файл myclock.c нам еще понадобится.

Компоновка

В предыдущем разделе говорилось о том, что компилятор переводит исходный код программы в исполняемый. Но это не всегда так.
В достаточно объемных программах исходный код обычно разделяется для удобства на несколько частей, которые компилируются отдельно, а затем соединяются воедино. Каждый такой "кусок" содержит *объектный код* и называется *объектным модулем*.
Объектные модули записываются в *объектные файлы*, имеющие расширение .o. В результате объединения объектных файлов могут получаться исполняемые файлы (обычные запускаемые бинарники), а также библиотеки.
Для объединения объектных файлов служит *компоновщик (линковщик)*, а сам процесс называют *компоновкой* или *линковкой*. В Linux имеется компоновщик GNU ld, входящий в состав пакета GNU binutils.

Ручная компоновка объектных файлов – довольно неприятный процесс, требующий передачи программе ld большого числа параметров, зависящих от многих фак-торов. К счастью, компиляторы из коллекции GCC сами вызывают линковщик с нужными параметрами, когда это необходимо.

Программист может самостоятельно передавать компоновщику дополнительные параметры через компилятор.

Теперь вернемся к нашему примеру. В предыдущем разделе компилятор "молча" вызвал компоновщик, в результате чего получился исполняемый файл.Чтобы отказаться от автоматической компоновки, нужно передать компилятору опцию -c:

$ gcc -c myclock.c

Если компилятор не нашел ошибок, то в текущем каталоге должен появиться объектный файл myclock.o. Других объектных файлов у нас нет, поэтому будем компоновать только его. Это делается очень просто:

$ gcc -o myclock myclock.o

В UNIX существуют различные форматы объектных файлов. Наиболее популярные среди них – a.out (Assembler OUTput) и COFF (Common Object File

Format). В Linux чаще всего встречается открытый формат объектных и исполняемых файлов ELF (Executable and Linkable Format).

Многофайловые проекты

Современные программные проекты редко ограничиваются одним исходным файлом. Распределение исходного кода программы на несколько файлов имеет ряд существенных преимуществ перед однофайловыми проектами.

Использование нескольких исходных файлов накладывает на *репозиторий* (рабочий каталог проекта) определенную логическую структуру. Такой код легче читать и модернизировать.

- В однофайловых проектах любая модернизация исходного кода влечет повтор-ную компиляцию всего проекта. В многофайловых проектах, напротив, доста-точно откомпилировать только измененный файл, чтобы обновить проект. Это экономит массу времени.
- Многофайловые проекты позволяют реализовывать одну программу на разных языках программирования.
- Многофайловые проекты позволяют применять к различным частям программы разные лицензионные соглашения.

Обычно процесс сборки многофайлового проекта осуществляется по следующему алгоритму:

1. Создаются и подготавливаются исходные файлы. Здесь есть одно важное замечание: каждый файл должен быть целостным, т. е. не должен содержать незавершенных конструкций. Функции и структуры не должны разрываться. Если в рамках проекта предполагается создание исполняемой программы, то в одном из исходных файлов должна присутствовать функция main().
2. Создаются и подготавливаются заголовочные файлы. У заголовочных файлов особая роль: они устанавливают соглашения по использованию общих идентификаторов (имен) в различных частях программы. Если, например, функция func() реализована в файле a.c, а вызывается в файле b.c, то в оба файла требуется включить директивой #include заголовочный файл, содержащий объявление (прототип) нашей функции. Технически можно обойтись и без заголовочных файлов, но в этом случае функцию можно будет вызвать с произвольными аргументами, и компилятор, за отсутствием соглашений, не выведет ни одной ошибки. Подобный "слепой" подход потенциально опасен и в большинстве случаев свидетельствует о плохом стиле программирования.
3. Каждый исходный файл отдельно компилируется с опцией -c. В результате появляется набор объектных файлов.
4. Полученные объектные файлы соединяются компоновщиком в одну исполняемую программу.

Если необходимо скомпоновать несколько объектных файлов (OBJ1.o, OBJ2.o и т. д.), то применяют следующий простой шаблон:

$ gcc -o OUTPUT_FILE OBJ1.o OBJ2.o ...

Рассмотрим программу, которая принимает в качестве аргумента строку и переводит в ней все символы в верхний регистр, т. е. заменяет все строчные буквы на заглавные. Чтобы не усложнять пример, будем преобразовывать только латинские (англоязычные) символы.
Для начала создадим файл print_up.h (листинг 1.2), в котором будет находиться объявление (прототип) функции print_up(). Эта функция переводит символы строки в верхний регистр и выводит полученный результат на экран.

```
void print_up (const char * str);
```

Листинг 1.2. Файл print_up.h

Итак, объявление функции print_up() устанавливает соглашение, по которому любой исходный файл, включающий print_up.h директивой #include, обязан вызвать функцию print_up() с одним и только одним аргументом типа const char*. Теперь создадим файл print_up.c (листинг 1.3), в котором будет находиться тело функции print_up().

```
#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include "print_up.h"
void print_up (const char * str)
{
int i;
for (i = 0; i < strlen (str); i++)
printf ("%c", toupper (str[i]));
printf ("\n");
}
```

Листинг 1.3. Файл print_up.c

Функция print_up() просматривает в цикле всю строку, посимвольно преобразовывая ее в верхний регистр. Вывод также производится посимвольно. В заключение выводится символ новой строки. Функция toupper(), объявленная в файле ctype.h и являющаяся частью стандартной библиотеки языка C, возвращает переданный ей символ в верхнем регистре, если это возможно. Без дополнительных манипуляций эта функция не работает с кириллическими (русскоязычными) символами, но сей-час это не важно. Теперь создадим третий файл main.c (листинг 1.4), который будет содержать функцию main(), необходимую для компоновки и запуска программы.

```
#include <string.h>
#include <stdio.h>
#include "print_up.h"
int main (int argc, char ** argv)
{
if (argc < 2) {
fprintf (stderr, "Wrong arguments\n");
```

```
return 1;
}
print_up (argv[1]);
return 0;
}
```
Листинг 1.4. Файл main.c

Сначала вспомним, что аргументы командной строки передаются в программу через функцию main():

- argc – целое число, содержащее количество аргументов командной строки;
- argv – массив строк (двумерный массив символов), в котором находятся аргу-менты.

Следует помнить, что в первый аргумент командной строки обычно заносится имя программы. Таким образом, argv[0] – это имя программы, argv[1] – первый переданный при запуске аргумент, argv[2] – второй аргумент и т. д. до элемента argv[argc-1].

Аргументы в программу можно передавать не только из командной оболочки. Поэтому понятие "аргументы командной строки" не всегда отражает действительное положение вещей. В связи с этим, чтобы избежать неоднозначности, будем в дальнейшем пользоваться более точной формулировкой "аргументы программы".

Теперь нужно собрать проект воедино. Сначала откомпилируем каждый файл с расширением .c:

```
$ gcc -c print_up.c
$ gcc -c main.c
```

В результате компиляции в репозитории программы должны появиться объектные файлы print_up.o и main.o, которые следует скомпоновать в один бинарный файл:

```
$ gcc -o printup print_up.o main.o
```

Если компилятор gcc вызывается с опцией -c, но без опции -o, то имя выходного файла получается заменой расширения .c на .o. Например, из файла foo.c получает-ся файл foo.o.

Осталось только запустить и протестировать программу:

```
$ ./printup
Wrong arguments
$ ./printup Hello
HELLO
```

Обратите внимание на то, что заголовочный файл print_up.h не компилируется. Заголовочные файлы вообще никогда отдельно не компилируются. Дело в том, что на стадии *препроцессирования* (условно первая стадия компиляции) все директивы #include заменяются на содержимое указанных в них файлов.

Собирать программы вручную неудобно, поэтому программисты, как правило, прибегают к различным приемам, позволяющим автоматизировать этот процесс. Самый простой способ – написать сценарий оболочки (shell-скрипт),

который будет автоматически выполнять все то, что вы обычно вводите вручную. Тогда многофайловый проект из предыдущей главы можно собрать, например, при помощи скрипта, приведенного в листинге 2.1.

```
#!/bin/sh
gcc -c print_up.c
gcc -c main.c
gcc -o printup print_up.o main.o
```

Листинг 2.1. Скрипт make_printup

Любая командная оболочка является также интерпретатором собственного языка программирования. В результате ей можно передавать набор команд в виде одного файла. Подобные файлы называются *скриптами (или сценариями) оболочки*. Каждый такой сценарий начинается с последовательности символов #! (решетка и восклицательный знак), после которой следует (без пробела) полный путь к исполняемому файлу оболочки, под которой будет выполняться скрипт. Такую строку называют shebang или hashbang. В Linux ссылка /bin/sh обычно указывает на оболочку bash.

Теперь файлу make_printup необходимо дать права на выполнение:

$ chmod +x make_printup

Осталось только вызвать скрипт, и проект будет создан:

$ ./make_printup

На первый взгляд, все прекрасно. Но настоящий программист должен предвидеть все возможные проблемы, и при детальном рассмотрении перспективы использования скрипта оболочки для сборки проекта уже не кажутся такими радужными. Перечислим некоторые проблемы.

- Скрипты оболочки статичны. Их работа не зависит от состояния текущей за- дачи. Даже если нужно заново откомпилировать только один файл, скрипт будет собирать проект "с нуля".
- В скриптах плохо просматриваются связи между различными элементами проекта.
- Скрипты не обладают возможностью самодиагностики.

К счастью, Linux имеет в наличии достаточно большой арсенал специализированных средств для автоматической сборки программных проектов. Такие средства называют *автосборщиками* или *утилитами автоматической сборки*. Благодаря специализированным автосборщикам программист может сосредоточитьсяна программировании, а не на процессе сборки.

3. Задание к лабораторной работе№2.

1. Ознакомиться с теоретической частью к лабораторной работе.
2. Используя редактор VI, создать текстовый файл, наполнить его содержимым. Добавить между первым и вторым словом файла фразу

«Люблю программировать». Удалить последнюю строку в файле. Сохранить изменения.

3. Используя редактор VI, написать программу myclock.c (Листинг 1.1). Откомпилировать программу, запустить на выполнение.
4. Создать многофайловый проект (листинги 1.2-1.4). Собрать проект воедино, откомпилировать каждый файл с расширением .c. Объектные файлы скомпоновать в один бинарный файл. Запустить и протестировать программу.
5. Многофайловый проект из п.4 собрать при помощи скрипта, приведенного в листинге 2.1. Запустить и протестировать программу.
6. Показать преподавателю исходные тексты программы на языке СИ, Продемонстрировать выполнение СИ – программы.
7. Удалить свои файлы и выйти из системы.

4. Контрольные вопросы

1. Для чего предназначен текстовый редактор VI? Основные функции редактора?
2. Какие режимы существуют в текстовом редакторе VI? Как осуществляется переключение между режимами?
3. Существует ли в редакторе VI история команд?
4. Как запустить программу в Linux на исполнение?
5. Как запустить многофайловый проект в linux на исполнение?
6. Что такое скрипт?
7. Что такое компановщик? Компилятор? Для чего они предназначены?

ЛАБОРАТОРНАЯ РАБОТА № 3. ИЗУЧЕНИЕ ФАЙЛОВОЙ СИСТЕМЫ И ФУНКЦИЙ ПО ОБРАБОТКЕ И УПРАВЛЕНИЮ ДАННЫМИ

1. Цель работы

Целью работы является изучение структуры файловой системы ОС LINUX, изучение команд создания, удаления, модификации файлов и каталогов, функций манипулирования данными.

2. Теоретическая часть

2.1. Файловая структура системы LINUX

В операционной системе LINUX файлами считаются обычные файлы, каталоги, а также специальные файлы, соответствующие периферийным устройствам (каждое устройство представляется в виде файла). Доступ ко всем файлам однотипный, в том числе, и к файлам периферийных устройств. Такой подход обеспечивает независимость программы пользователя от особенностей ввода/вывода на конкретное внешнее устройство.

Файловая структура LINUX имеет иерархическую древовидную структуру. В корневом каталоге размещаются другие каталоги и файлы, включая 5 основных каталогов:

bin – большинство выполняемых командных программ и *shell* – процедур;

tmp – временные файлы;

usr – каталоги пользователей (условное обозначение);

etc – преимущественно административные утилиты и файлы;

dev – специальные файлы, представляющие периферийные устройства; при добавлении периферийного устройства в каталог /dev должен быть добавлен соответствующий файл (черта / означает принадлежность корневому каталогу).

Текущий каталог – это каталог, в котором в данный момент находится пользователь. При наличии прав доступа, пользователь может перейти после входа в систему в другой каталог. Текущий каталог обозначается точкой (**.**); родительский каталог, которому принадлежит текущий, обозначается двумя точками (**..**).

Полное имя файла может включать имена каталогов, включая корневой, разделенных косой чертой, например: /home/student/file.txt. Первая косая черта

обозначает корневой каталог, и поиск файла будет начинаться с него, а затем в каталоге home, затем в каталоге student.

Один файл можно сделать принадлежащим нескольким каталогам. Для этого используется команда **ln (link)**:

ln<имя файла 1><имя файла 2>

Имя 1-го файла – это полное составное имя файла, с которым устанавливается связь; имя 2-го файла – это полное имя файла в новом каталоге, где будет использоваться эта связь. Новое имя может не отличаться от старого. Каждый файл может иметь несколько связей, т.е. он может использоваться в разных каталогах под разными именами. Команда **ln** с аргументом -s создает символическую связь:

ln -s <имя файла 1><имя файла 2>

Здесь имя 2-го файла является именем символической связи. Символическая связь является особым видом файла, в котором хранится имя файла, на который символическая связь ссылается. LINUX работает с символической связью не так, как с обычным файлом – например, при выводе на экран содержимого символической связи появятся данные файла, на который эта символическая связь ссылается.

В LINUX различаются 3 уровня доступа к файлам и каталогам:

1) доступ владельца файла;

2) доступ группы пользователей, к которой принадлежит владелец файла;

3) остальные пользователи.

Для каждого уровня существуют свои байты атрибутов, значение которых расшифровывается следующим образом:

r – разрешение на чтение;

w – разрешение на запись;

x – разрешение на выполнение;

- – отсутствие разрешения.

Первый символ байта атрибутов определяет тип файла и может интерпретироваться со следующими значениями:

- – обычный файл;

d – каталог;

l – символическая связь;

в – блок-ориентированный специальный файл, который соответствует таким периферийным устройствам, как накопители на магнитных дисках;

с – байт-ориентированный специальный файл, который может соответствовать таким периферийным устройствам как принтер, терминал.

В домашнем каталоге пользователь имеет полный доступ к файлам (READ, WRITE, EXECUTE; r, w, x).

Атрибуты файла можно просмотреть командой **ls -l** и они представляются в следующем формате:

d rwxrwxrwx

| | | |

| | | | Доступ для остальных пользователей

| | | Доступ к файлу для членов группы

| | Доступ к файлу владельца

| Тип файла (директория)

Пример. Командой **ls -l** получим листинг содержимого текущей директории student:

- rwx --- --- 2 student 100 Mar 10 10:30 file_1

- rwx --- r-- 1 adm 200 May 20 11:15 file_2

- rwx --- r-- 1 student 100 May 20 12:50 file_3

После байтов атрибутов на экран выводится следующая информация о файле:

- число связей файла;

- имя владельца файла;

- размер файла в байтах;

- дата создания файла (или модификации);

- время;

- имя файла.

Атрибуты файла и доступ к нему, можно изменить командой:

chmod<коды защиты><имя файла>

Коды защиты могут быть заданы в числовом или символьном виде. Для символьного кода используются:

знак плюс (+) – добавить права доступа;

знак минус (-) – отменить права доступа;

r,w,x – доступ на чтение, запись, выполнение;

u,g,o – владельца, группы, остальных.

Коды защиты в числовом виде могут быть заданы в восьмеричной форме. Для контроля установленного доступа к своему файлу после каждого изменения кода защиты нужно проверять свои действия с помощью команды **ls -l.**

Примеры:

chmod g+rw,o+r file.1 – установка атрибутов чтения и записи для группы и чтения для всех остальных пользователей;

ls -l file.1 – чтение атрибутов файла;

chmod o-w file.1 – отмена атрибута записи у остальных пользователей;

>letter – создание файла letter. Символ > используется как для переадресации, так и для создания файла;

cat – вывод содержимого файла;

cat file.1 file.2 > file.12 – конкатенация файлов (объединение);

mv file.1 file.2 – переименование файла file.1 в file.2;

mv file.1 file.2 file.3 directory – перемещениефайлов file.1, file.2, file.3 вуказаннуюдиректорию;

rm file.1 file.2 file.3 – удалениефайлов file.1, file.2, file.3;.

cp file.1 file.2 – копирование файла с переименованием;

mkdirnamedir – создание каталога;

rm dir_1 dir_2 – удаление каталогов dir_1 dir_2;

ls [acdfgilqrstv CFR] namedir – вывод содержимого каталога; если в качестве namedir указано имя файла, то выдается вся информация об этом файле. Значения аргументов:

- l –- список включает всю информацию о файлах;

- t – сортировка по времени модификации файлов;

- a – в список включаются все файлы, в том числе и те, которые начинаются с точки;

- s – размеры файлов указываются в блоках;

- d – вывести имя самого каталога, но не содержимое;

- r – сортировка строк вывода;

- i – указать идентификационный номер каждого файла;

- v – сортировка файлов по времени последнего доступа;

- q – непечатаемые символы заменить на знак ?;

- c – использовать время создания файла при сортировке;

- g – то же что -l, но с указанием имени группы пользователей;

- f – вывод содержимого всех указанных каталогов, отменяет флаги -l, -t, -s, -r и активизирует флаг -a;

- C – вывод элементов каталога в несколько столбцов;

- F – добавление к имени каталога символа / и символа * к имени файла, для которых разрешено выполнение;

- R – рекурсивный вывод содержимого подкаталогов заданного каталога.

cd<namedir> – переход в другой каталог. Если параметры не указаны, то происходит переход в домашний каталог пользователя.

pwd – вывод имени текущего каталога;

grep [-vcilns] [шаблон поиска] <имя файла> – поиск файлов с указанием или без указания контекста (шаблона поиска).

Значение ключей:

- v – выводятся строки, не содержащие шаблон поиска;

- c – выводится только число строк, содержащих или не содержащих шаблон;

- i – при поиске не различаются прописные и строчные буквы;

- l – выводятся только имена файлов, содержащие указанный шаблон;

- n – перенумеровать выводимые строки;

- s – формируется только код завершения.

Примеры.

1. Напечатать имена всех файлов текущего каталога, содержащих последовательность "student" и имеющих расширение .txt:

grep -lstudent *.txt

2. Определить имя пользователя, входящего в ОС LINUX с терминала tty23:

who | grep tty23

3. Задание к лабораторной работе№3.

1. Ознакомиться с файловой структурой ОС LINUX. Изучить команды работы с файлами.
2. Используя команды ОС LINUX, создать два текстовых файла.
3. Полученные файлы объединить в один файл и его содержимое просмотреть на экране.
4. Создать новую директорию и переместить в нее полученные файлы.
5. Вывести полную информацию обо всех файлах и проанализировать уровни доступа.
6. Добавить для всех трех файлов право выполнения членам группы и остальным пользователям.
7. Просмотреть атрибуты файлов.
8. Создать еще один каталог.

9. Установить дополнительную связь объединенного файла с новым каталогом, но под другим именем.
10. Создать символическую связь.
11. Сделать текущим новый каталог и вывести на экран расширенный список информации о его файлах.
12. Произвести поиск заданной последовательности символов в файлах текущей директории и получить перечень соответствующих файлов.
13. Получить информацию об активных процессах и имена других пользователей.
14. Сдать отчет о работе и удалить свои файлы и каталоги.
15. Выйти из системы.

4. Контрольные вопросы

1. Что считается файлами в ОС LINUX?
2. Объясните назначение связей с файлами и способы их создания.
3. Что определяет атрибуты файлов и каким образом их можно просмотреть и изменить?
4. Какие методы создания и удаления файлов, каталогов Вы знаете?
5. В чем заключается поиск по шаблону?
6. Какой командой можно получить список работающих пользователей и сохранить его в файле?

ЛАБОРАТОРНАЯ РАБОТА № 4. СОЗДАНИЕ И ВЫПОЛНЕНИЕ КОМАНДНЫХ ФАЙЛОВ В СРЕДЕ ОС LINUX.

1. Цель работы

Целью работы является изучение методов создания и выполнения командных файлов на языке Shell – интерпретатора.

2. Теоретическая часть

В предыдущих лабораторных работах взаимодействие с командным интерпретатором Shell осуществлялось с помощью командной строки. Однако,Shell является также и языком программирования, который применяется для написания командных файлов (shell – файлов). Командные файлы также называются скриптами и сценариями. Shell – файл содержит одну или несколько выполняемых команд (процедур), а имя файла в этом случае используется как имя команды.

2.1. Переменные командного интерпретатора

Для обозначения переменных Shell используется последовательность букв, цифр и символов подчеркивания; переменные не могут начинаться с цифры. Присваивание значений переменным проводится с использованием знака = , например, PS2 = '< . Для обращения к значению переменной перед ее именем ставится знак $. Их можно разделить на следующие группы:

- позиционные переменные вида $n, где n – целое число;

- простые переменные, значения которых может задавать пользователь или они могут устанавливаться интерпретатором;

- специальные переменные # ? – ! $ устанавливаются интерпретатором и позволяют получить информацию о числе позиционных переменных, коде завершения последней команды, идентификационном номере текущего и фонового процессов, о текущих флагах интерпретатора Shell.

Простые переменные. Shell присваивает значения переменным:

```
z=1000

x= $z

echo $x

1000
```

Здесь переменной x присвоено значение z.

Позиционные переменные. Переменные вида $n, где n – целое число, используются для идентификации позиций элементов в командной строке с помощью номеров, начиная с нуля. Например, в командной строке

cat text_1 text_2...text_9

аргументы идентифицируются параметрами $1...$9. Для имени команды всегда используется S0. В данном случае $0 – это cat, $1 – text_1, $2 – text_2 и т.д. Для присваивания значений позиционным переменным используется команда **set**, например:

set arg_1 arg_2... arg_9

здесь $1 присваивается значение аргумента arg_1, $2 – arg_2 и т.д.

Для доступа к аргументам используется команда **echo**, например:

echo $1 $2 $9

arg_1 arg_2 arg_9

Для получения информации обо всех аргументах (включая последний) используют метасимвол *. Пример:

echo $*

arg_2 arg_3 ... arg_10 arg_11 arg_12

С помощью позиционных переменных Shell можно сохранить имя команды и ее аргументы. При выполнении команды интерпретатор Shell должен передать ей аргументы, порядок которых может регулироваться также с помощью позиционных переменных.

Специальные переменные. Переменные – ? # $! устанавливаются только Shell. Они позволяют с помощью команды **echo** получить следующую информацию:

- – текущие флаги интерпретатора (установка флагов может быть изменена командой **set**);

– число аргументов, которое было сохранено интерпретатором при выполнении какой-либо команды;

? – код возврата последней выполняемой команды;

$ – числовой идентификатор текущего процесса PID;

! – PID последнего фонового процесса.

2.2. Арифметические операции

Команда **expr** (express -- выражать) вычисляет выражение expression и записывает результат в стандартный вывод. Элементы выражения разделяются пробелами; символы, имеющие специальный смысл в командном языке, нужно экранировать. Строки, содержащие специальные символы, заключают в апострофы. Используя команду **expr**, можно выполнять сложение, вычитание, умножение, деление, взятие остатка, сопоставление символов и т. д.

Пример. Сложение, вычитание:

b=190

a=` expr 200 – $b`

где ` – обратная кавычка (левая верхняя клавиша). Умножение *, деление /, взятие остатка %:

d=` expr $a + 125 "*" 10`

c=` expr $d % 13`

Здесь знак умножения заключается в двойные кавычки, чтобы интерпретатор не воспринимал его как метасимвол. Во второй строке переменной *c* присваивается значение остатка от деления переменной *d* на 13.

Сопоставление символов с указанием числа совпадающих символов:

concur=` expr "abcdefgh" : "abcde"`

echo $concur

ответ 5.

Операция сопоставления обозначается двоеточием (:). Результат – переменная concur.

Подсчет числа символов в цепочках символов. Операция выполняется с использованием функции *length* в команде **expr**:

chain="The program is written in Assembler"

str=` expr length "$chain"`

Echo $str

ответ 35. Здесь результат подсчета обозначен переменной *str*.

2.3. Встроенные команды

Встроенные команды являются частью интерпретатора и не требуют для своего выполнения проведения последовательного поиска файла команды и создания новых процессов. Встроенные команды:

cd [dir] – назначение текущего каталога;

exec [cmd [arg...]]<имя файла> – выполнение команды, заданной аргументами cmd и arg, путем вызова соответствующего выполняемого файла.

umask[-o | -s] [nnn] – устанавливает маску создания файла (маску режимов доступа создаваемого файла, равную восьмеричному числу nnn: 3 восьмеричных цифры для пользователя, группы и других). Если аргумент nnn отсутствует, то команда сообщает текущее значение маски. При наличии флага -o маска выводится в восьмеричном виде, при наличии флага -s – в символьном представлении;

set, unset – режим работы интерпретатора, присваивание значений параметрам;

eval[-arg] – вычисление и выполнение команды;

sh<filename.sh> выполнение командного файла filename.sh;

exit [n] – приводит к прекращению выполнения программы, возвращает код возврата, равный нулю, в вызывающую программу;

trap [cmd] [cond] – перехват сигналов прерывания, где: cmd – выполняемая команда; cond=0 или EXIT – в этом случае команда cmdвыполняется при завершении интерпретатора; cond=ERR – команда cmd выполняется при обнаружении ошибки; cond – символьное или числовое обозначение сигнала, в этом случае команда cmd выполняется при приходе этого сигнала;

export [name [=word]...] – включение в среду. Команда **export** объявляет, что переменные name будут включаться в среду всех вызываемых впоследствии команд;

wait [n] – ожидание завершения процесса. Команда без аргументов ожидает завершения процессов, запущенных синхронно. Если указан числовой аргумент n, то **wait** ожидает фоновый процесс с номером n;

readname – команда вводит строку со стандартного ввода и присваивает прочитанные слова переменным, заданным аргументами name.

Пример. Пусть имеется shell-файл *data*, содержащий две команды:

echo -n "Please write down your name:"

readname

Если вызвать файл на выполнение, введя его имя, то на экране появится сообщение:

Please write down your name:

Программа ожидает ввода с клавиатуры (в данном случае – фамилии пользователя). После ввода фамилии и нажатия клавиши *Enter* команда выполнится и на следующей строке появится знак – приглашение.

2.4. Управление программами

Команды **true** и **false** служат для установления требуемого кода завершения процесса: true – успешное завершение, код завершения 0; false – неуспешное завершение, код может иметь несколько значений, с помощью которых определяется причина неуспешного завершения. Коды завершения команд используются для принятия решения о дальнейших действиях в операторах цикла **while** и **until** и в условном операторе **if**. Многие команды LINUX вырабатывают код завершения только для поддержки этих операторов.

Условный оператор if проверяет значение выражения. Если оно равно true, Shell выполняет следующий за **if** оператор, если false, то следующий оператор пропускается. Формат оператора **if:**

if<условие>

then

list1

else

list2

fi

Команда **test** (проверить) используется с условным оператором **if** и операторами циклов. Действия при этом зависят от кода возврата **test. Test** проводит анализ файлов, числовых значений, цепочек символов. Нулевой код

выдается, если при проверке результат положителен, ненулевой код при отрицательном результате проверки.

В случае анализа файлов синтаксис команды следующий:

test[-rwfds] file

где

-r – файл существует и его можно прочитать (код завершения 0);

-w – файл существует и в него можно записывать;

-f – файл существует и не является каталогом;

-d – файл существует и является каталогом;

-s – размер файла отличен от нуля.

При анализе числовых значений команда **test** проверяет, истинно ли данное отношение, например, равны ли A и B . Сравнение выполняется в формате:

-eq A = B

-ne A<>B

test A -ge B эквивалентно A>= B

-le A<= B

-gt A> B

-lt A< B

Отношения слева используются для числовых данных, справа – для символов.

Кроме команды **test** имеются еще некоторые средства для проверки:

! – операция отрицания инвертирует значение выражения, например, выражение **iftesttrue** эквивалентно выражению **iftest !false**;

o – двуместная операция "ИЛИ" (or) дает значение true, если один из операндов имеет значение true;

a – двуместная операция "И" (and) дает значение true, если оба операнда имеют значение true.

2.5. Циклы

Оператор цикла с условием **while**true и **while**false. Команда **while** (пока) формирует циклы, которые выполняются до тех пор, пока команда **while** определяет значение следующего за ним выражения как true или false. Форматоператорацикласусловием**while** true:

while list1

do

list2

done

Здесь list1 и list2 – списки команд. **While** проверяет код возврата списка команд, стоящих после **while**, и если его значение равно 0, то выполняются команды, стоящие между **do** и **done**. Оператор цикла с условием **while**false имеет формат:

until list1

do

list2

done

В отличие от предыдущего случая условием выполнения команд между **do** и **done** является ненулевое значение возврата. Программный цикл может быть размещен внутри другого цикла (вложенный цикл). Оператор **break** прерывает ближайший к нему цикл. Если в программу ввести оператор **break** с уровнем 2 (**break 2**), то это обеспечит выход за пределы двух циклов и завершение программы.

Оператор **continue** передает управление ближайшему в цикле оператору **while**.

Оператор цикла с перечислением **for**:

for name in [wordlist]

do

list

done

где name – переменная; wordlist – последовательность слов; list – список команд. Переменная name получает значение первого слова последовательности wordlist, после этого выполняется список команд, стоящий между **do** и **done**. Затем name получает значение второго слова wordlist и снова выполняется список list. Выполнение прекращается после того, как кончится список wordlist.

Ветвление по многим направлениям **case**. Команда **case** обеспечивает ветвление по многим направлениям в зависимости от значений аргументов команды. Формат:

case<string> in

s1) <list1>;;

s2) <list2>;;

.

.

.

sn) <listn>;;

***) <list>**

esac

Здесь list1, list2 ... listn – список команд. Производится сравнение шаблона string с шаблонами s1, s2 ... sk ... sn. При совпадении выполняется список команд, стоящий между текущим шаблоном sk и соответствующими знаками ;;. Пример:

echo -n 'Please, write down your age'

read age

case $age in

test $age -le 20) echo 'you are so young' ;;

test $age -le 40) echo 'you are still young' ;;

test $age -le 70) echo 'you are too young' ;;

***)echo 'Please, write down once more'**

esac

В конце текста помещена звездочка * на случай неправильного ввода числа.

3. Задание к лабораторной работе№4.

Составьте и выполните shell – программы, включающей следующие действия:

1. Вывод на экран списка параметров командной строки с указанием номера каждого параметра.
2. Присвоение переменным A, B и C значений 10, 100 и 200, вычисление и вывод результатов по формуле D=(A*2 + B/3)*C.
3. Формирование файла со списком файлов в домашнем каталоге, вывод на экран этого списка в алфавитном порядке и общего количества файлов.
4. Переход в другой каталог, формирование файла с листингом каталога и возвращение в исходный каталог.
5. Запрос и ввод имени пользователя, сравнение с текущим логическим именем пользователя и вывод сообщения: верно/неверно.
6. Запрос и ввод имени файла в текущем каталоге и вывод сообщения о типе файла.
7. Циклическое чтение системного времени и очистка экрана в заданный момент.
8. Циклический просмотр списка файлов и выдача сообщения при появлении заданного имени в списке.

4. Контрольные вопросы

1. Какое назначение имеют shell – файлы?
2. Как создать shell – файл и сделать его выполняемым?
3. Какие типы переменных используются в shell – файлах?
4. В чем заключается анализ цепочки символов?
5. Какие встроенные команды используются в shell – файлах?
6. Как производится управление программами?
7. Назовите операторы создания циклов.

ЛАБОРАТОРНАЯ РАБОТА № 5. ИЗУЧЕНИЕ ГРАФИЧЕСКОЙ ОБОЛОЧКИ KDE

1. Цель работы

Целью работы является изучение работы с основными функциональными частями графической оболочки KDE, получение навыков по настройке KDE и создании простейших текстовых и графических документов в KWord и Paint.

2. Общие теоретические сведения

K Desktop Environment (Среда рабочего стола K) KDE предназначена для поддержания тех же функциональных возможностей графического интерфейса, какие предоставляют и другие популярные системы, например MacOS и Windows. Кроме выполнения стандартных функций, KDE обладает рядом специфических характеристик, которые расширяют возможности графической среды. Для Linuxразработано несколько диспетчеров окон, таких, как *olwm, fvwm, afterstep* и другие. Однако, их возможности не идут ни в какое сравнение с возможностями KDE.

2.1 Оконная среда KDE

Как и большинство оконных менеджеров, KDE представляет собой интегрированную среду, содержащую базовые средства для решения ряда повседневных задач. Например, с помощью **KDE** можно выполнять ряд операций:

- Размещение на рабочем столе ярлыков гибких дисков для их монтирования, размонтирования и работы с ними.
- Отображение в графическом виде файловой структуры и перемещение по ней.
- Сопоставление приложений с файлами определенных типов. При этом если щелкнуть на выбранном файле, автоматически будет загружаться нужное приложение.
- Создание на рабочем столе ярлыков принтеров. Если мышью перетащить к такому ярлыку файл, он будет распечатан.

В состав KDE входит не только рабочий стол, но и целый набор приложений и утилит для работы с ним. В стандартном дистрибутиве KDE имеется более сотни программ – от игр и системных утилит до целых блоков офисных программ. Кроме того, приложения KDE могут взаимодействовать друг с другом для упрощения выполнения всевозможных операций.

2.2 Компоненты рабочего стола KDE.

Рабочий стол KDE разделен на три основные части – «поверхность» рабочего стола, панель и линейку задач. Основная рабочая область среды KDE называется рабочим столом. Это тот фон, на котором отображаются все другие компоненты. На рабочем столе можно размещать ярлыки программ, документов и устройств, к которым чаще всего приходится обращаться. Это позволяет легко получать доступ к соответствующим объектам для работы с ними. Кроме той области, что отображается на экране, KDE предоставляет дополнительное виртуальное рабочее пространство для выполнения программ. По умолчанию поддерживается четыре виртуальных рабочих стола. Виртуальный рабочий стол – это, по сути, другой экран, на который можно переключиться для того, чтобы запустить приложение или выполнить еще какую-то работу. Программы и окна легко перемещаются между различными виртуальными рабочими столами. Дополнительные возможности, предоставляемые за счет использования виртуальных рабочих столов, могут быть использованы самыми разными программами. При этом нет необходимости сворачивать и разворачивать окна выполняемых приложений. Можно просто отложить выполняемое приложение в таком виде, как есть, а затем вернуться к нему по завершении выполнения.

2.2.1 Панель

Панель располагается в нижней части экрана. На панели размещаются кнопки, позволяющие выполнять основные процедуры KDE, а также ярлыки наиболее часто используемых программ. Одним из особо важных элементов на панели является кнопка Application Starter (Запуск Приложений), которая расположена (по умолчанию) в левой части панели. Это кнопка с литерой "К" над изображением зубчатого колеса. С ее помощью можно открыть меню, в котором представлены все приложения, установленные на данную систему. Кроме того, это же меню может быть использовано для доступа к некоторым другим разделам KDE, таким, как диалоговая справка и *Панель Управления (Control Panel).*

На панели размещен переключатель виртуальных рабочих столов *Пейджер, Панель Задач (Taskbar)* и *Часы (Clock).* Панель задач отображает открытые на текущем рабочем столе окна. Чтобы получить немедленный доступ к программе, нужно просто щелкнуть в соответствующем месте на панели задач.

Запустить на выполнение программу можно одним из перечисленных ниже способов.

• **Щелкнуть кнопкой на панели.** Некоторые программы представлены по умолчанию на панели в виде ярлыков или кнопок, например эмулятор виртуальных рабочих столов, панель управления, вызов справки и текстовой редактор.

• **Щелкнуть на элементе рабочего стола.** По умолчанию на рабочем столе размещается только два объекта. Это *Корзина* и ярлык рабочего каталога. Пользователи сами размещают на рабочем столе наиболее нужные и часто используемые программы.

• **Выбрать программу из меню запуска приложений.** Достаточно щелкнуть на литере "К" и выбрать тот пункт меню, который соответствует запускаемому приложению.

• **Использовать диспетчер файлов.** В окне диспетчера файлов нужно выбрать соответствующий файл и щелкнуть на нем мышью.

Можно, конечно, запустить программу на выполнение в командной строке окна терминала – задать название программы. Можно также нажатием клавиш <Alt+F2> вызвать окно запуска программ и ввести туда название программы.

Ряд полезных программ облегчает работу пользователя.

В первую очередь, это программа эмуляции терминала konsole, позволяющая открывать окна и получать доступ к стандартной командной строке. На панели имеется соответствующая кнопка с изображением маленького монитора и ракушки.

Справку в диалоговом режиме можно получить, если щелкнуть на кнопке панели с изображением спасательного круга. Справка включает в себя разные темы, как, например, программа-гид для начинающих пользователей и система контекстного поиска для используемых в KDE приложений.

Просмотреть файловую систему или получить доступ к ресурсам World Wide Web можно, используя окно диспетчера файлов. Для того чтобы диспетчер файлов отобразил в своем окне содержимое рабочего каталога, нужно щелкнуть на папке панели с изображением домика.

Щелканье по кнопке ► удаляет панель с экрана. Эта кнопка остается при этом на экране, так что можно вернуть панель обратно. Это свойство действует только на открытый в данный момент рабочий стол; другие рабочие столы сохраняют вид мини – или главной панели.

Список задач – кнопка, расположенная справа от меню приложений (обозначена пиктограммой монитора), несет меню, содержащее все активные на данный момент окна, отсортированные по имени. Это позволяет легко и быстро найти необходимое окно и уменьшает захламленность экрана при работе с несколькими окнами.

2.2.2 Настройка KDE

Центр управления (Control Center) (кнопка с изображением гаечного ключа) составляет основу всей системы настроек KDE. В ее входит множество панелей для всевозможных компонентов рабочей среды и даже некоторых приложений KDE.

В центре управления используется деление на группы, щелкнув на знаке "плюс" в углу группы, можно увидеть список входящих в группу компонентов. Щелкнув на знаке "минус" в том же углу группы, этот список можно свернуть. Доступ к любому диалогу с раскрытым деревом меню можно получить при помощи Preferences (Предпочтения) и из меню запуска программ Start Application.

Большинство диалоговых окон имеют кнопку вызова справки. В самом простом случае это контекстная справка. Для ее получения нужно щелкнуть мышью на знаке вопроса на рамке окна. Курсор мыши при этом изменит свой вид на стрелку с большим знаком вопроса. Если теперь щелкнуть на том элементе диалогового окна, с которым возникли трудности, появится прямоугольник желтого цвета с текстом справки. Для получения более детальной справки можно воспользоваться опцией Help (Справка) на левой панели. Наконец, если возникли проблемы с поиском необходимого диалогового окна, на этой же панели нужно выбрать опцию Search (Поиск). Затем нужно ввести ключевое слово, по которому и будет осуществляться поиск.

Control Center

KDE предоставляет широкие возможности по модифицированию внешнего вида окон и рабочей области, включая отображение фона, ярлыков, шрифтов и тому подобное. Не представляет труда и управление работой отдельных компонентов, вроде того же рабочего стола или окна. Например, можно управлять реакцией элемента на щелчок мышью, процессом загрузки и отображения выбранных окон, выбирать хранитель экрана. Все эти и многие другие возможности может предоставить рассматриваемая группа Control Center.

Для настройки параметров работы рабочего стола и окон следует выбрать опцию нужного диалога настройки под названием Desktop на дереве центра управления.

Изменение схемы цветов

Диалоговое окно выбора цвета Appearance&Themes $\Rightarrow$ Colors (Цвета) предназначено для изменения используемой цветовой схемы для окон KDE и других графических приложений.

Цветовая схема включает в себя 18 пунктов выбора цвета для различных элементов окна программы и установки контрастов. В области предварительного просмотра отображаются все элементы окна, реагирующие на изменение цветовой схемы. Как только пользователь меняет параметры или установки, в области просмотра отражаются внесенные изменения. Можно выбрать уже готовую цветовую схему из списка Color Scheme (Схема Цветов).

Для изменения какой-то конкретной установки нужно выбрать соответствующий элемент из выпадающего меню области цветов Widget Color (Декорация) или щелкнуть в нужной части окна предварительного просмотра. После того как элемент выбран, можно изменить его цвет. Для этого достаточно щелкнуть на кнопке и выбрать понравившийся цвет из появившегося диалогового окна выбора цвета.

Контраст изменяется при помощи позиционирования специального рычажка контраста, который может размещаться в диапазоне от Low (Низкий) до High (Высокий). Эти установки применяются при отображении трехмерных рамок вокруг элементов интерфейса приложений KDE.

Для подтверждения выбора следует щелкнуть на кнопке Apply (Применить). Если приходится часто менять цветовые установки, бывает полезно внести изменения в список цветовых схем. Для этого нужно щелкнуть на кнопке Save Scheme и задать название для своей схемы. Для удаления схемы из списка нужно выделить ее и щелкнуть на кнопке Remove (Удалить).

Изменение фона

Для изменения цвета фона или фонового узора рабочего стола нужно на дереве опций центра управления последовательно выбрать Control Center=>Appearance&Themes=>Background. В результате появится диалоговое окно, имеющее три основные области:

• список виртуальных рабочих столов;

• окно предварительного просмотра;

• окно настройки параметров.

Каждый виртуальный стол в KDE имеет собственные настройки фона. Для каждого такого стола можно выбрать фон с одноцветной или двухцветной палитрой, а также фоновый узор. Если используется фоновый узор, можно задать способ его отображения. Можно также выбрать несколько узоров и автоматически переключаться между ними. Доступны и более усовершенствованные опции, позволяющие сочетать цвета и узоры, а также поддерживать динамические настройки фона.

В процессе внесения изменений в установки они отображаются в окне предварительного просмотра.

Виртуальные рабочие столы

Производить настройку параметров виртуальных рабочих столов в KDE можно в диалоговом окне Control Center ⇒Desktop ⇒Multiple Desktops.

Указатель Number of Desktops (Количество рабочих столов) показывает, сколько виртуальных рабочих столов доступно. Их число может изменяться в диапазоне от одного до шестнадцати. Здесь же можно задать название для рабочего стола, которое потом будет отображено в списке окон (Window List) или использовано в настройках панели.

Хранитель экрана

Диалоговое окно выбора хранителя экрана Appearance&Themes ⇒ Screensaver позволяет выбрать хранитель экрана и осуществить настройку его параметров. Опции настройки бывают глобальные, как, например, опция установки времени запуска хранителя экрана, и индивидуальные – для каждого отдельного хранителя. Диалоговое окно выбора хранителя экрана имеет три основные секции:

• окно предварительного просмотра;

• список программ – хранителей экрана;

• опции настройки.

Необходимо выбрать название нужной программы из предложенного списка. Для настройки параметров необходимо щелкнуть на кнопке Setup (Настройка) и в появившемся диалоговом окне произвести установку нужных характеристик.

Для установки интервала времени, через который будет запускаться хранитель экрана, нужно ввести в поле опции Settings (Установки) величину данного интервала в минутах.

Параметр Priority (Приоритет) позволяет определить распределение процессором времени на работу хранителя экрана. Это пример того, как в Linux организована многозадачность. Если нужно, чтобы у хранителя был наивысший приоритет (например, для качественного вывода анимации), следует передвинуть рычажок в позицию High (Высокий). Если же, наоборот, необходимо обеспечить высокий приоритет других процессов, нужная позиция для рычажка приоритетности хранителя – Low (Низкий).

Для того чтобы проверить выполненные установки, следует щелкнуть на кнопке Test (Просмотр). Для подтверждения сделанного выбора нажмите кнопку OK или Apply.

Настройка диспетчера окон

С помощью опций Control Center Appearance&Themes ⇒ Desktop/Window behavior (Поведение Окон) центра управления можно устанавливать поведение диспетчера окон. Эти настройки определяют способ отображения окон в случае их перемещения и изменения размера, а также управляют процессом разворачивания, размещения и выделения окон при работе с диспетчером окон. Опции в верхней части диалогового окна позволяют выполнить настройку параметров, задающих режим перемещения окна и изменения его размеров, а также определяют функциональность команды Maximize (Развернуть). Можно задать такой режим отображения окна при перемещении или изменении его размера, что окно будет отображаться вместе со всем своим содержимым или же в виде прозрачной рамки. Если выбран режим отображения всего содержимого окна, процесс перемещения или изменения размеров окна будет требовать дополнительного времени для обновления отображаемых на экране элементов.

Если используются окна с изменяемыми размерами, можно выбрать режим обновления содержимого окна при каждом изменении его размера. Для этого следует воспользоваться установками Resize(Изменение размера). Для выбора частоты обновления можно воспользоваться специальным рычажком. Если сделан выбор, отличный от None (Никакой), то каждый раз, при изменении размеров окна, его содержимое будет обновляться. Это дает возможность отслеживать процесс заполнения окна программой и позволяет выбрать оптимальные размеры последнего.

Window behavior/Moving – меню установок размещения окна на экране Placement (Расположение) позволяет определить место на экране, где будет отображаться окно. Поддерживаются такие методы.

• **Smart (Умный)** – минимизируется перекрытие между окнами.

• **Cascade (Каскад)** – первое окно отображается в левом верхнем углу. Следующее окно отображается сдвинутым немного вправо и вниз, так что окна практически полностью перекрываются. И так далее. Окна расположены, как карты в руке при игре в преферанс.

• **Random (Произвольный)** – окна располагаются на экране в произвольном порядке.

Метод получения фокуса (т.е. метод выделения отдельных окон или элементов) является, пожалуй, индивидуальным методом настроек KDE. С помощью этого метода определяется, какое из открытых окон активно и какие следует выполнить действия при активизации окна. Более детально это выглядит так.

• **Click to focus (Передача фокуса щелчком).** Окно получает фокус (т.е. становится активным) при щелчке на нем мышью. При этом окно автоматически выводится на первый план по отношению к другим окнам. Такой метод используется по умолчанию.

• **Focus follows mouse (Фокус за мышью).** Окно получает фокус при непосредственном обращении к нему (это можно сделать с помощью указателя мыши, используя комбинацию клавиш <Alt+Tab> и тому подобное). При этом окно может подниматься поверх других окон, а может и не подниматься. Перемещение указателя мыши на рабочую область за пределы окна не означает потерю последним фокуса. При выборе опции Auto Raise (Всплывать автоматически) окно будет всплывать на экране при перемещении в его область курсора в течение нескольких миллисекунд. Число этих миллисекунд устанавливается с помощью рычажка Delay (Задержка). Если выбрана опция Click Raise (Всплывать при щелчке), окно будет подниматься поверх других окон при щелчке в любой части окна. В противном случае такая реакция окна будет наблюдаться только при щелчке на его заголовке. Это исключительно полезный метод передачи фокуса, поскольку позволяет набирать текст в одном окне и одновременно читать содержимое другого окна, расположенного частично поверх указанного.

• **Focus Under Mouse (Фокус под мышью).** Окно получает фокус при любом перемещении на него указателя мыши. При этом комбинация клавиш <Alt+Tab> может и не помочь.

• **Focus Strictly Under Mouse (Фокус только под мышью).** Окно получает фокус, только если указатель мыши находится внутри окна. Если указатель мыши находится в рабочей области, где нет окон, ни одно из окон не получит фокус.

Использование окон

Открытое окно состоит из следующих элементов.

Window menu (Меню управления окном) – В левом верхнем углу каждого окна находится пиктограмма манипулирования окном. При щелчке на ней появляется меню, содержащее команды с помощью которых можно манипулировать данным окном. Maximize (Максимизировать) увеличит окно до максимально возможного размера. Minimize (Минимизировать) сделает ваше окно невидимым. Move(Переместить) позволяет передвигать окно с помощью

мыши. Size (Изменить размер) позволит вам увеличить или уменьшить окно. Shade – свернет окно до заголовка. To desktop...(На рабочий стол) позволит перевести окно на другой рабочий стол. Выберите рабочий стол, на который вы хотите переместить это окно. Окно при этом исчезнет. Для того чтобы увидеть его снова, выберите имя на Линейке задач, или щелкните на соответствующую кнопку рабочего стола на панели KDE. Close (Закрыть) закроет данное окно. Always on Top – оставляет окно поверх всех открытых окон.

Использование панели меню каждого окна в KDE очень просто. Щелкните на команду, и она будет исполнена. При нажатии на правую кнопку мыши появится контекстное меню, позволяющее вывести на экран панель меню. Можете отсоединить меню от окна и оставить его "плавать" по экрану.

Ниже панели меню находятся пиктограммы инструментов, которые позволяют исполнять различные команды. Можно передвинуть инструментальную панель – влево, вправо, вверх, вниз, и, конечно, она тоже может "плавать".

3. Задание к лабораторной работе№5.

1. Запустите Центр управлений.
2. Поменяйте Фон, сначала на одноцветный, а затем вставьте фоновое изображение.
3. Установите хранитель экрана, на своё усмотрение, и режим ожидания равный минуте.
4. Сделайте так, чтобы окна передвигались вместе со всем их содержимым.
5. Задайте звуковой щелчок, подтверждающий нажатие каждой клавиши.
6. Измените ширину линейки панели.
7. Запустите диспетчер приложений. И запустите программу текстового процессора KWord.
8. В другом рабочем столе откройте программу растрового редактора Paint.
9. Откройте KWord и наберите следующий текст:

   The Quick Brown Fox Jumps Over The Lazy Dog, используя два разных стиля по вашему выбору. Сохраните этот файл в домашнем каталоге пользователя, закройте KWord.

10. Откройте ваш домашний каталог пользователя Konqueror'ом, создайте в нем каталог, скопируйте ваш текстовой файл в этот каталог.
11. Ознакомьтесь с содержанием домашнего каталога, скопируйте с дискеты файлы.
12. Получите справку об интересующем вас объекте.

13. Создайте любой рисунок с помощью Paint, чтобы в нем были ВСЕ фигуры (1. эллипс, 2. окружность, 3. линия, 4. прямоугольник, 5. круг) хотя бы по одному разу и присутствовало не менее четырех цветов.
14. Сохраните файл с рисунком в домашнем каталоге, закройте Paint.
15. Скопируйте файл с рисунком в тот же созданный вами каталог.
16. Измените атрибуты доступа к созданным файлам.
17. Покажите преподавателю ваши файлы, затем удалите их.

4. Контрольные вопросы

1. Перечислите стандартные функции KDE.
2. Что является компонентом рабочего стола KDE?
3. Назовите функции панели рабочего стола.
4. Как получить справку в диалоговом режиме?
5. Какие функции предоставляет центр управления KDE?

ЛАБОРАТОРНАЯ РАБОТА № 6. ПРОЦЕССЫ В ОПЕРАЦИОННОЙ СИСТЕМЕ LINUX.

1. Цель работы

Целью работы является изучение процессов в ОС Linux, знакомство с системными вызовами getppid() и getpid(),fork(),семейством функций для системного вызова exec().

2. Теоретическая часть

2.1. Понятие процесса в Linux. Его контекст

Все построение операционной системы UNIX основано на использовании концепции процессов, которая обсуждалась на лекции. Контекст процесса складывается из пользовательского контекста и контекста ядра, как изображено на рис.1.

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);
- неинициализируемые 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 С функций malloc(), calloc(), realloc()).

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

Рисунок 2 – Контекст процесса в LINUX

Под понятием «контекст ядра» объединяются системный контекст и регистровый контекст, рассмотренные на лекции. Мы будем выделять в контексте ядрастек ядра, который используется при работе процесса в режиме ядра (kernelmode), 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.

2.2. Идентификация процесса

Каждый процесс в операционной системе получает уникальный идентификационный номер – PID (processidentificator) . 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 2^{31}-1.

2.3. Состояния процесса. Краткая диаграмма состояний

Модель состояний процессов в операционной системе LINUX представляет собой детализацию модели состояний, принятой в лекционном курсе. Краткая диаграмма состояний процессов в операционной системе LINUX изображена на рис.2.

Рисунок 3 – Сокращенная диаграмма состояний процесса в LINUX

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.

Приведенная выше диаграмма состояний процессов в LINUX не является полной. Она показывает только состояния, для понимания которых достаточно уже полученных знаний.

2.4. Иерархия процессов

В операционной системе LINUX все процессы, кроме одного, создающегося при старте операционной системы, могут быть порождены только какими-либо другими процессами. В качестве прародителя всех остальных процессов в

подобных LINUX системах могут выступать процессы с номерами 1 или 0. В операционной системе Linux таким родоначальником, существующим только при загрузке системы, является процесс kernel с идентификатором 0.

Таким образом, все процессы в LINUX связаны отношениями процесс-родитель – процесс-ребенок и образуют генеалогическое дерево процессов. Для сохранения целостности генеалогического дерева в ситуациях, когда процесс-родитель завершает свою работу до завершения выполнения процесса-ребенка, идентификатор родительского процесса в данных ядра процесса-ребенка (PPID – parentprocessidentificator) изменяет свое значение на значение1, соответствующее идентификатору процесса init, время жизни которого определяет время функционирования операционной системы. Тем самым процесс init как бы усыновляет осиротевшие процессы. Наверное, логичнее было бы заменять PPID не на значение1, а на значение идентификатора ближайшего существующего процесса-прародителя умершего процесса-родителя, но в LINUX почему-то такая схема реализована не была.

2.5. 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(). Прототипы этих системных вызовов и соответствующие типы данных описаны в системных файлах <sys/types.h> и <unistd.h>. 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_tgetpid(void);
pid_tgetppid(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 С.

2.6. Создание процесса в LINUX. Системный вызов fork()

В операционной системе LINUX новый процесс может быть порожден единственным способом – с помощью системного вызова fork(). При этом вновь созданный процесс будет являться практически полной копией родительского процесса. 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, о чем подробнее будет рассказано на семинарах, когда мы будем говорить о сигналах в операционной системе LINUX.

Системный вызов для порождения нового процесса

Прототип системного вызова

```
#include<sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

Описание системного вызова

Системный вызов fork служит для создания нового процесса в операционной системе LINUX. Процесс, который инициировал системный вызов fork, принято называть родительским процессом (parentprocess). Вновь порожденный процесс принято называть процессом-ребенком (childprocess). Процесс-ребенок является почти полной копией родительского процесса. У порожденного процесса по сравнению с родительским изменяются значения следующих параметров:

- идентификатор процесса;
- идентификатор родительского процесса;
- время, оставшееся до получения сигнала SIGALRM ;
- сигналы, ожидавшие доставки родительскому процессу, не будут доставляться порожденному процессу.

При однократном системном вызове возврат из него может произойти дважды: один раз в родительском процессе, а второй раз в порожденном процессе. Если создание нового процесса произошло успешно, то в порожденном процессе системный вызов вернет значение 0, а в родительском процессе – положительное значение, равное идентификатору процесса-ребенка.

Если создать новый процесс не удалось, то системный вызов вернет в инициировавший его процесс отрицательное значение.

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.

2.7. Создание программы с fork() с одинаковой работой родителя и ребенка

Для иллюстрации сказанного давайте рассмотрим следующую программу:

```
/* Программа primer6-1.c – пример создания нового
 процесса с одинаковой работой процессов
 ребенка и родителя */

#include<sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
pid_tpid, 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 роди-
тельского процесса (в каждом из
      процессов !!!) */

pid = getpid();
```

```
ppid = getppid();

   /* Печатаем значения PID, PPID и вычислен-
ное 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 .6.1. Программа primer6-1.c – пример создания нового процесса с одинаковой работой процессов ребенка и родителя.

Для того чтобы после возвращения из системного вызова fork()процессы могли определить, кто из них является ребенком, а кто родителем, и, соответственно, по-разному организовать свое поведение, системный вызов возвращает в них разные значения. При успешном создании нового процесса в процесс-родитель возвращается положительное значение, равное идентификатору процесса-ребенка. В процесс-ребенок же возвращается значение0. Если по какой-либо причине создать новый процесс не удалось, то системный вызов вернет в инициировавший его процесс значение-1. Таким образом, общая схема организации различной работы процесса-ребенка и процесса-родителя выглядит так:

```
pid = fork();
if(pid == -1){
   ...
   /* ошибка */
...
} else if (pid == 0){
   ...
   /* ребенок */
   ...
} else {
...
   /* родитель */
   ...
}
```

2.8. Завершение процесса. Функция exit()

Существует два способа корректного завершения процесса в программах, написанных на языке С. 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 функцияexit()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);
```

Описание функции

Функция exitслужит для нормального завершения процесса. При выполнении этой функции происходит сброс всех частично заполненных буферов ввода-вывода с закрытием соответствующих потоков (файлов, pipe, FIFO, сокетов), 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 status – кода завершения процесса – передается ядру операционной системы и может быть затем получено процессом, породившим завершившийся процесс. При этом используются только младшие 8 бит параметра, так что для кода завершения допустимы значения от 0 до 255. По соглашению, код завершения 0 означает безошибочное завершение процесса.

Если процесс завершает свою работу раньше, чем его родитель, и родитель явно не указал, что он не хочет получать информацию о статусе завершения порожденного процесса, то завершившийся процесс не исчезает из системы окончательно, а остается в состоянии **закончил исполнение** либо до завершения процесса-родителя, либо до того момента, когда родитель получит эту информацию. Процессы, находящиеся в состоянии **закончил исполнение**, в операционной системе LINUX принято называть процессами-зомби (zombie, defunct).

2.9. Параметры функции main() в языке С. Переменные среды и аргументы командной строки

У функции main() в языке программирования С существует три параметра, которые могут быть переданы ей операционной системой. Полный прототип функцииmain() выглядит следующим образом:

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

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, компиляторgcc, 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 LINUX. 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 значениепеременной 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 указательNULL.

2.10. 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 рис.3

Рисунок 4 – Взаимосвязь различных функций для выполнения системного вызова exec()

Функции изменения пользовательского контекста процесса

Прототипы функций

```
#include <unistd.h>
int execlp(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 execle(const char *path,
   const char *arg0,
... const char *argN,(char *)NULL,
   char * envp[])
int execve(const char *path, char *argv[],
char *envp[])
```

Описание функций

Для загрузки новой программы в системный контекст текущего процесса используется семейство взаимосвязанных функций, отличающихся друг от друга формой представления параметров.

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:

- идентификатор процесса;
- идентификатор родительского процесса;
- групповой идентификатор процесса;
- идентификатор сеанса;
- время, оставшееся до возникновения сигнала SIGALRM ;
- текущую рабочую директорию;
- маску создания файлов;
- идентификатор пользователя;
- групповой идентификатор пользователя;
- явное игнорирование сигналов;

- таблицу открытых файлов (если для файлового дескриптора не устанавливался признак "закрыть файл при выполнении exec()").

В случае успешного выполнения возврата из функций в программу, осуществившую вызов, не происходит, а управление передается загруженной программе. В случае неудачного выполнения в программу, инициировавшую вызов, возвращается отрицательное значение.

Поскольку системный контекст процесса при вызове exec() остается практически неизменным, большинство атрибутов процесса, доступных пользователю через системные вызовы (PID, UID, GID, PPID и другие, смысл которых станет понятен по мере углубления наших знаний на дальнейших занятиях), 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**.

2.11. Создание программы с использованием системного вызова exec()

Для иллюстрации использования системного вызова exec()давайте рассмотрим следующую программу

```
/* Программа primer6-2.c, изменяющая пользователь-
ский контекст процесса (запускающая
  другуюпрограмму) */

#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 execle 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
   хотимраспечатать. */

(void) execle("/bin/cat", "/bin/cat",
"03-2.c", 0, envp);

/* Сюда попадаем только при
   возникновенииошибки */
printf("Error on program start\n");
exit(-1);
return 0;     /* Никогда не выполняется, нужен
                 для того, чтобы компилятор не
                 выдавал warning */
}
```

Листинг 6.2. Программа primer6-2.c, изменяющая пользовательский контекст процесса

3. Задание к лабораторной работе№6.

1. Ознакомиться с теоретической частью к лабораторной работе.
2. Наберите программу primer6-1.c, откомпилируйте ее и запустите на исполнение (лучше всего это делать не из оболочки mc, так как она не очень корректно сбрасывает буферы ввода-вывода). Проанализируйте полученный результат.
3. Измените предыдущую программу с fork()так, чтобы родитель и ребенок совершали разные действия (какие – не важно).
4. Откомпилируйте программу primer2.c и запустите на исполнение. Поскольку при нормальной работе будет распечатываться содержимое файла с именем primer6-2.c, такой файл при запуске должен присутствовать в текущей директории (проще всего записать исходный текст программы под этим именем). Проанализируйте результат.
5. (*)В качестве примера самостоятельно напишите программу, распечатывающую значения аргументов командной строки и параметров окружающей среды для текущего процесса.
6. (*)Для закрепления полученных знаний модифицируйте программу, созданную при выполнении задания3 так, чтобы порожденный процесс запускал на исполнение новую (любую) программу.

4. Контрольные вопросы

1. Что понимается под процессом в ОС Linux?
2. Что такое пользовательский контекст? Контекст ядра?

3. В чем отличие классической (пятикомпонентной) модели состояния процесса от модели состояний процессов в ОС Linux (рис.2)?
4. С помощью какого системного вызова может быть получено значение идентификатора текущего процесса, значение идентификатора родительского процесса для текущего процесса?
5. Каким способом в ОС Linux может быть порожден новый процесс? Яляется ли данный способ единственным?
6. Для чего применяется системный вызов exec()? Может ли пользователь вызвать его непосредственно?
7. Какими функциями может воспользоваться программист для осуществления вызова exec()?
8. В чем разница между системными вызовами fork() и exec()?

ЛАБОРАТОРНАЯ РАБОТА № 7. ОРГАНИЗАЦИЯ ВЗАИМОДЕЙСТВИЯ ПРОЦЕССОВ ЧЕРЕЗ PIPE И FIFO В LINUX

1. Цель работы

Целью работы является изучение организации взаимодействия процессов, получение представления о работе с файлами через системные вызовы и стандартную библиотеку ввода-вывода. Понятие файлового дескриптора. Организация связи через pipe между процессом-родителем и процессом-потомком. Понятие FIFO. Использование системного вызова mknod() для создания FIFO.

2. Теоретическая часть

2.1 Понятие о потоке ввода-вывода

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 потоковая 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.

2.2 Понятие о работе с файлами через системные вызовы и стандартную библиотеку ввода-вывода для языка С

Потоковая передача информации может осуществляться не только между процессами, но и между процессом и устройством ввода-вывода, например между процессом и диском, на котором данные представляются в виде файла. Поскольку понятие файла должно быть знакомо изучающим этот курс, а системные вызовы, использующиеся для потоковой работы с файлом, во многом соответствуют системным вызовам, применяемым для потокового общения процессов, мы начнем наше рассмотрение именно с механизма потокового обмена между процессом и файлом.

Из курса программирования на языке С вам известны функции работы с файлами из стандартной библиотеки ввода-вывода, такие как fopen(), fread(), fwrite(), fprintf(), fscanf(), fgets() и т.д. Эти функции входят как неотъемлемая часть в стандарт ANSI на язык С и позволяют программисту получать информацию из файла или записывать ее в файл при условии, что программист обладает определенными знаниями о содержимом передаваемых данных. Так, например, функция fgets() используется для ввода из файла последовательности

символов, заканчивающейся символом '\n' – перевод каретки. Функция fscanf() производит ввод информации, соответствующей заданному формату, и т. д. С точки зрения потоковой модели операции, определяемые функциями стандартной библиотеки ввода-вывода, не являются потоковыми операциями, так как каждая из них требует наличия некоторой структуры передаваемых данных.

В операционной системе LINUX эти функции представляют собой надстройку – сервисный интерфейс – над системными вызовами, осуществляющими прямые потоковые операции обмена информацией между процессом и файлом и не требующими никаких знаний о том, что она содержит.

2.3 Файловый дескриптор

На лекции мы говорили, что информация о файлах, используемых процессом, входит в состав его системного контекста и хранится в его блоке управления – PCB. В операционной системе LINUX можно упрощенно полагать, что информация о файлах, с которыми процесс осуществляет операции потокового обмена, наряду с информацией о потоковых линиях связи, соединяющих процесс с другими процессами и устройствами ввода-вывода, хранится в некотором массиве, получившем название таблицы открытых файлов или таблицы файловых дескрипторов. Индекс элемента этого массива, соответствующий определенному потоку ввода-вывода, получил название файлового дескриптора для этого потока. Таким образом, файловый дескриптор представляет собой небольшое целое неотрицательное число, которое для текущего процесса в данный момент времени однозначно определяет некоторый действующий канал ввода-вывода. Некоторые файловые дескрипторы на этапе старта любой программы ассоциируются со стандартными потоками ввода-вывода. Так, например, файловый дескриптор 0 соответствует стандартному потоку ввода, файловый дескриптор 1 – стандартному потоку вывода, файловый дескриптор 2 – стандартному потоку для вывода ошибок. В нормальном интерактивном режиме работы стандартный поток ввода связывает процесс с клавиатурой, а стандартные потоки вывода и вывода ошибок – с текущим терминалом.

2.4 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, мы должны поместить информацию о файле в таблицу открытых файлов и определить соответствующий файловый дескриптор. Для этого применяется процедура открытия файла, осуществляемая системным вызовом 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 LINUX 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 соответсвующий 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 процессаumask, 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

Системный вызовopen()использует набор флагов для того, чтобы специфицировать операции, которые предполагается применять к файлу в дальнейшем или которые должны быть выполнены непосредственно в момент открытия файла. Из всего возможного набора флагов на текущем уровне знаний нас будут интересовать только флаги O_RDONLY, O_WRONLY, O_RDWR, O_CREAT и O_EXCL. Первые три флага являются взаимоисключающими: хотя бы один из них должен быть применен и наличие одного из них не допускает наличия двух других. Эти флаги описывают набор операций, которые, при успешном открытии файла, будут разрешены над файлом в дальнейшем: только чтение, только запись, чтение и запись.У каждого файла существуют атрибуты прав доступа для различных категорий пользователей. Если файл с заданным именем существует на диске, и права доступа к нему для пользователя, от имени которого работает текущий процесс, не противоречат запрошенному набору операций, то операционная система сканирует таблицу открытых файлов от ее начала к концу в поисках первого свободного элемента, заполняет его и возвращает индекс этого элемента в качестве файлового дескриптора открытого файла. Если файла на диске нет, не хватает прав или отсутствует свободное место в таблице открытых файлов, то констатируется возникновение ошибки.

В случае, когда мы **допускаем**, что файл на диске может отсутствовать, и хотим, чтобы он был создан, флаг для набора операций должен использоваться в комбинации с флагом O_CREAT. Если файл существует, то все происходит по рассмотренному выше сценарию. Если файла нет, сначала выполняется создание файла с набором прав, указанным в параметрах системного вызова. Проверка соответствия набора операций объявленным правам доступа может и не производиться (как, например, в Linux).

В случае, когда мы **требуем**, чтобы файл на диске отсутствовал и был создан в момент открытия, флаг для набора операций должен использоваться в комбинации с флагами O_CREAT и O_EXCL.

2.5 Системные вызовы read(), write(), close()

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

Системные вызовы read и write

Прототипы системных вызовов

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

size_tnbytes);

Описание системных вызовов

Системные вызовы readи 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. Заметим, что это значение (большее 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.

Мы сейчас не акцентируем внимание на понятии указателя текущей позиции в файле и взаимном влиянии значения этого указателя и поведения системных вызовов.

После завершения потоковых операций процесс должен выполнить операцию закрытия потока ввода-вывода, во время которой произойдет окончательный сброс буферов на линии связи, освободятся выделенные ресурсы операционной системы, и элемент таблицы открытых файлов, соответствующий файловому дескриптору, будет отмечен как свободный. За эти действия отвечает системный вызов close(). Надо отметить, что при завершении работы процесса с помощью явного или неявного вызова функции exit() происходит автоматическое закрытие всех открытых потоков ввода-вывода.

Системный вызов close

Прототип системного вызова

```
#include <unistd.h>
int close(intfd);
```

Описание системного вызова

Системный вызов closeпредназначен 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.

2.6 Создание программы для записи информации в файл

Для иллюстрации сказанного давайте рассмотрим следующую программу:

```
/*Программа Primer7-1.c, иллюстрирующая использование системных вызовов
open(), write() и close() для записи информации в файл */
#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
int main(){
intfd;
size_t size;
charstring[] = "Hello, world!";
   /* Обнуляем маску создания файлов текущего процесса для того,
   чтобы права доступа у создаваемого файла точно соответствовали
```

```
   параметру вызова open() */
   (void)umask(0);
   /* Попытаемся открыть файл с именем myfile в текущей директории
    только для операций вывода. Если файла не существует, попробуем
    его создать 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, печатаем об этом сообщение
      ипрекращаемработу */
printf("Can\'t open file\n");
exit(-1);
}
   /* Пробуем записать 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 != 14){
      /* Если записалось меньшее количество байт, сообщаем об
      ошибке */
printf("Can\'t write all string\n");
exit(-1);
   }
   /* Закрываемфайл */
if(close(fd) < 0){
printf("Can\'t close file\n");
}
return 0;
}
```

Листинг 7.1. Программа Primer 7-1.c, иллюстрирующая использование системных вызовов open(), write() и close() для записи информации в файл

2.7 Понятие о pipe. Системный вызов pipe()

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

Важное отличие pip'a 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 перегнать входной и наоборот. Для создания нового экземпляра такого кольцевого буфера внутри операционной системы используется системный вызовpipe ().

Прототип системного вызова

```
#include <unistd.h>
int pipe(int *fd);
```

Описание системного вызова

Системный вызов pipe предназначен для создания pip'a 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 pip’a 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’a, 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’a в системе не может превышать время жизни процессов, работающих с ним.

2.8 Создание программы для pipe в одном процессе

Достаточно яркой иллюстрацией действий по созданию pip'a, записи в него данных, чтению из него и освобождению выделенных ресурсов может служить программа, организующая работу с pip’ом в рамках одного процесса, приведенная ниже:

```
/* Программа Primer 7-2.c, иллюстрирующая работу с pip'ом в рамках одного
процесса */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(){
intfd[2];
size_t size;
char string[] = "Hello, world!";
charresstring[14];
/* Попытаемся создать pipe */
if(pipe(fd) < 0){
      /* Если создать pipe не удалось, печатаем об этом сообщение
      и прекращаем работу */
printf("Can\'t createpipe\n");
exit(-1);
   }
   /* Пробуем записать в pipe 14 байт из нашего 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, сообщаем об
      ошибке */
printf("Can\'t write all string\n");
exit(-1);
   }
   /* Пробуем прочитать из pip'a 14 байт в другой массив, т.е. всю
   записаннуюстроку */
   size = read(fd[0], resstring, 14);
   if(size < 0){
/* Если прочитать не смогли, сообщаем об ошибке */
printf("Can\'t read string\n");
exit(-1);
   }
   /* Печатаем прочитанную строку */
printf("%s\n",resstring);
/* Закрываемвходнойпоток*/
if(close(fd[0]) < 0){
```

```
printf("Can\'t close input stream\n");
}
   /* Закрываемвыходнойпоток*/
if(close(fd[1]) < 0){
printf("Can\'t close output stream\n");
}
return 0;
}
```

Листинг 7.2. Программа Primer 7-2.c, иллюстрирующая работу с pip'ом в рамках одного процесса

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

Понятно, что если бы все достоинство pip'ов сводилось к замене функции копирования из памяти в память внутри одного процесса на пересылку информации через операционную систему, то его использование было бы неоптимальным. Однако, 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(), однако их рассмотрение выходит за рамки нашего курса). Это обстоятельство позволяет организовать передачу информации через pipe между родственными процессами, имеющими общего прародителя, создавшего pipe.

2.10 Создание программы для организации однонаправленной связи между родственными процессами через pipe

Рассмотрим программу, осуществляющую однонаправленную связь между процессом-родителем и процессом-ребенком:

```
/* Программа Primer 7-3.c, осуществляющая однонаправленную связь через pipe
между процессом-родителем и процессом-ребенком */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(){
intfd[2], result;
size_t size;
charresstring[14];
   /* Попытаемся создать pipe */
if(pipe(fd) < 0){
```

```
        /* Если создать pipe не удалось, печатаем об этом сообщение
        и прекращаем работу */
printf("Can\'t createpipe\n");
exit(-1);
    }
    /* Порождаем новый процесс */
result = fork();
if(result){
        /* Если создать процесс не удалось, сообщаем об этом и
        завершаемработу */
printf("Can\'t fork child\n");
exit(-1);
    } elseif (result> 0) {
        /* Мы находимся в родительском процессе, который будет
        передавать информацию процессу-ребенку. В этом процессе
        выходной поток данных нам не понадобится, поэтому
        закрываем его.*/
close(fd[0]);
        /* Пробуем записать 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, сообщаем
            об ошибке и завершаем работу */
printf("Can\'t write all string\n");
exit(-1);
        }
        /* Закрываем входной поток данных, на этом
        родитель прекращает работу */
close(fd[1]);
printf("Parent exit\n");
    } else {
/* Мы находимся в порожденном процессе, который будет
        получать информацию от процесса-родителя. Он унаследовал
        от родителя таблицу открытых файлов и, зная файловые
        дескрипторы, соответствующие pip, иможет его использовать.
        В этом процессе входной поток данных нам не
ипонадобится, поэтому закрываем его.*/
close(fd[1]);
        /* Пробуем прочитать из pip'a 14 байт в массив, т.е. всю
        записаннуюстроку */
        size = read(fd[0], resstring, 14);
        if(size < 0){

/* Если прочитать не смогли, сообщаем об ошибке и
```

```
        завершаемработу */

printf("Can\'t read string\n");
exit(-1);
      }
      /* Печатаем прочитанную строку */
printf("%s\n",resstring);
      /* Закрываем входной поток и завершаем работу */
close(fd[0]);
   }
return 0;
}
```

Листинг 7.3. Программа Primer 7-3.c, осуществляющая однонаправленную связь через pipe между процессом-родителем и процессом-ребенком

Будьте внимательны при написании программ, обменивающихся большими объемами информации через pipe. **Помните, что 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. Проверяйте значения, возвращаемые вызовами**!

2.11 Понятие FIFO. Использование системного вызова mknod() для создания FIFO. Функция mkfifo()

Доступ к информации о расположении pip'a в операционной системе и его состоянии может быть осуществлен только через таблицу открытых файлов процесса, создавшего pipe, и через унаследованные от него таблицы открытых файлов процессов-потомков. Поэтому изложенный выше механизм обмена информацией через pipe справедлив лишь для родственных процессов, имеющих общего прародителя, инициировавшего системный вызов pipe(), или для таких процессов и самого прародителя и не может использоваться для потокового общения с другими процессами. В операционной системе LINUX существует возможность использования pip'a для взаимодействия других процессов, но ее реализация достаточно сложна и лежит за пределами наших занятий.

Для организации потокового взаимодействия любых процессов в операционной системе LINUX применяется средство связи, получившее название FIFO (от FirstInputFirstOutput) или именованный pipe. FIFO во всем подобен pip'у, за одним исключением: данные о расположении FIFO в адресном пространстве ядра и его состоянии процессы могут получать не через родственные связи, а через файловую систему. Для этого при создании именованного pip'a 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 LINUX 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 pip'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>
intmknod(char *path, int mode, int dev);
```

Описание системного вызова

Нашей целью является не полное описание системного вызова mknod, а только описание его использования для создания FIFO. Поэтому мы будем рассматривать не все возможные варианты задания параметров, а только те из них, которые соответствуют этой специфической деятельности.

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 процессаumask, 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>
intmkfifo(char *path, int mode);

Описание функции

Функция mkfifoпредназначена 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 процессаumask, а именно – они равны (0777 &mode) & ~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 MidnightCommander (mc)!!! Это приведет к его глубокому зависанию!

2.12 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 вызовopen()при открытии FIFO также 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 значениефайлового 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.

2.13 Создание программы с FIFO в родственных процессах

Для иллюстрации взаимодействия процессов через FIFO рассмотрим такую программу:

```
/* Программа Primer 7-4.c, 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>
int main(){
intfd, result;
size_t size;
   char resstring[14];
   char name[]="aaa.fifo";
/* Обнуляем маску создания файлов текущего процесса для того,
   чтобы права доступа у создаваемого FIFO точно соответствовали
   параметру вызова mknod() */
   (void)umask(0);
   /* Попытаемся создать FIFO с именем aaa.fifo в текущей
   директории */
   if(mknod(name, S_IFIFO | 0666, 0) < 0){
/* Если создать FIFO не удалось, печатаем об этом
      сообщение и прекращаем работу */
printf("Can\'t create FIFO\n");
exit(-1);
   }
   /* Порождаем новый процесс */
if((result = fork()) < 0){
      /* Если создать процесс не удалось, сообщаем об этом и
      завершаемработу */
printf("Can\'t fork child\n");
exit(-1);
   } elseif (result> 0) {
      /* Мы находимся в родительском процессе, который будет
      передавать информацию процессу-ребенку. В этом процессе
      открываем FIFO на запись.*/
if((fd = open(name, O_WRONLY)) < 0){
         /* Если открыть FIFO не удалось, печатаем об этом
         сообщение и прекращаем работу */
printf("Can\'t open FIFO forwriting\n");
exit(-1);
      }
      /* Пробуем 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,то сообщаем
      об ошибке и завершаем работу */
printf("Can\'t write all string to FIFO\n");
exit(-1);
      }
      /* Закрываем входной поток данных и на этом родитель
```

```
        прекращаетработу */
        close(fd);
printf("Parent exit\n");
} else {
        /* Мы находимся в порожденном процессе, который будет
        получать информацию от процесса-родителя. Открываем
        FIFO на чтение.*/
if((fd = open(name, O_RDONLY)) < 0){
            /* Если открыть FIFO не удалось, печатаем об этом
            сообщениеипрекращаемработу */
printf("Can\'t open FIFO for reading\n");
            exit(-1);
}
        /* Пробуем прочитать из FIFO 14 байт в массив, т.е.
        всю записанную строку */
size = read(fd, resstring, 14);
if(size< 0){
            /* Если прочитать не смогли, сообщаем об ошибке
            изавершаемработу */
printf("Can\'t read string\n");
exit(-1);
        }
        /* Печатаем прочитанную строку */
printf("%s\n",resstring);
        /* Закрываем входной поток и завершаем работу */
close(fd);
    }
return 0;
}
```

Листинг 7.4. Программа Primer 7-4.c, осуществляющая однонаправленную связь через FIFO между процессом-родителем и процессом-ребенком

3. Задание к лабораторной работе№7.

1. Ознакомиться с теоретической частью к лабораторной работе.
2. Наберите, откомпилируйте программу Primer 7-1.c и запустите ее на исполнение. 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. Измените программу из предыдущего пункта задания так, чтобы она читала записанную ранее в файл информацию и печатала ее на экране. Все лишние операторы желательно удалить.
4. Наберите программу Primer 7-2.c, откомпилируйте ее и запустите на исполнение.

5. Наберите программу Primer 7-3.c, откомпилируйте ее и запустите на исполнение.
6. (*)Модифицируйте пример из п.5 для связи между собой двух родственных процессов, исполняющих разные программы.
7. (*)Определите размер pipe для вашей операционной системы.
8. Наберите программу Primer 7-4.c, откомпилируйте ее и запустите на исполнение. В этой программе информацией между собой обмениваются процесс-родитель и процесс-ребенок. Обратим внимание, что 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 mknod().
9. (*)Для закрепления полученных знаний напишите на базе предыдущего примера (п.8) 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).

4. Контрольные вопросы

1. Какие модели передачи данных по каналам связи вам известны?
2. Что представляет собой файловый дескриптор?
3. Чему соответствует файловый дескриптор 0? файловый дескриптор 1? файловый дескриптор 2?
4. Для чего применяется системный вызов open()?
5. Какие вызовы применяются для совершения потоковых операций чтения информации из файла и ее записи в файл?
6. За какие действия отвечает системный вызовclose()?
7. Что такое pipe?
8. В чем заключается важное отличие pip’a от файла?
9. В чем разница между FIFO и pipe?
10. Какие особенности поведения имеют системные вызовы read() и write() при работе с FIFO?

ЛАБОРАТОРНАЯ РАБОТА № 8. СЕМАФОРЫ В LINUX КАК СРЕДСТВО СИНХРОНИЗАЦИИ ПРОЦЕССОВ

1. Цель работы

Целью работы является изучение семафоров в ОС Отличие операций над LINUX-семафорами от классических операций. Создание массива семафоров или доступ к уже существующему массиву. Выполнение операций над семафорами.

2. Теоретическая часть

2.3 Семафоры в LINUX. Отличие операций над LINUX-семафорами от классических операций

Как упоминалось на лекционных занятиях, 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. Следует отметить, что набор операций над семафорами System V IPC отличается от классического набора операций {P, V}, предложенного Дейкстрой. Он включает три операции:

- A(S, n)– увеличить значение семафораS на величину n ;
- D(S, n)– пока значение семафораS < n, процесс блокируется. Далее S = S – n;
- Z(S)– процесс блокируется до тех пор, пока значение семафораS не станет равным 0 .

Изначально все IPC-семафоры инициируются нулевым значением.

Мы видим, что классической операцииP(S) соответствует операция D(S,1), а классической операцииV(S) соответствует операция A(S,1). Аналогом ненулевой инициализации семафоров Дейкстры значением n может служить выполнение операцииA(S,n)сразу после создания семафора S, с обеспечением атомарности создания семафора и ее выполнения посредством другого семафора. Мы показали, что классические семафоры реализуются через семафоры System V IPC. Обратное не является верным. Используя операцииP(S) и V(S), мы не сумеем 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 вызовам в качестве параметра передаются IPC- 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.

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

В целях экономии системных ресурсов операционная система LINUX 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 доступа по ключу к уже существующему массиву используется системный вызов semget(), являющийся аналогом системного вызова shmget() для разделяемой памяти, который возвращает значение IPC-дескриптора для этого массива. При этом применяются те же способы создания и доступа, что и для разделяемой памяти. Вновь созданные семафоры инициируются нулевым значением.

Системный вызов semget()

Прототип системного вызова

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

intsemget(key_t key, intnsems,
intsemflg);
```

Описание системного вызова

Системный вызов semgetпредназначен для выполнения операции доступа к массиву IPC-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 инициируются нулевым значением.

Возвращаемое значение

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

2.5 Выполнение операций над семафорами. Системный вызов semop()

Для выполнения операций A, D и Z над семафорами из массива используется системный вызов semop(), обладающий довольно сложной семантикой. Разработчики System V IPC явно перегрузили этот вызов,

применяя его не только для выполнения всех трех операций, но еще и для нескольких семафоров в массиве IPC-семафоров одновременно. Для правильного использования этого вызова необходимо выполнить следующие действия:

1. Определиться, для каких семафоров из массива предстоит выполнить операции. Необходимо иметь в виду, что все операции реально совершаются только перед успешным возвращением из системного вызова, т.е. если вы хотите выполнить операции A(S1,5)и Z(S2)в одном вызове и оказалось, что S2 != 0, то значение семафораS1 не будет изменено до тех пор, пока значение S2 не станет равным 0. Порядок выполнения операций в случае, когда процесс не переходит в состояние **ожидание**, не определен. Так, например, при одновременном выполнении операций A(S1,1)и D(S2,1)в случае S2 > 1 неизвестно, что произойдет раньше – уменьшится значение семафораS2 или увеличится значение семафораS1. Если порядок для вас важен, лучше применить несколько вызовов вместо одного.
2. После того как вы определились с количеством семафоров и совершаемыми операциями, необходимо завести в программе массив из элементов типа structsembuf с размерностью, равной определенному количеству семафоров (если операция совершается только над одним семафором, можно, естественно, обойтись просто переменной). Каждый элемент этого массива будет соответствовать операции над одним семафором.
3. Заполнить элементы массива. В поле sem_flg каждого элемента нужно занести значение 0 (другие значения флагов в семинарах мы рассматривать не будем). В поля sem_num и sem_op следует занести номера семафоров в массиве IPC семафоров и соответствующие коды операций. Семафоры нумеруются, начиная с 0. Если у вас в массиве всего один семафор, то он будет иметь номер 0. Операции кодируются так:

   - для выполнения операции A(S,n)значение поля sem_op должно быть равно n ;
   - для выполнения операции D(S,n)значение поля sem_op должно быть равно –n ;
   - для выполнения операции Z(S)значение поля sem_op должно быть равно 0.

4. В качестве второго параметра системного вызова semop()указать адрес заполненного массива, а в качестве третьего параметра – ранее определенное количество семафоров, над которыми совершаются операции.

Системный вызов semop()

Прототип системного вызова

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

intsemop(intsemid, structsembuf *sops,
intnsops);
```

Описание системного вызова

Системный вызов semop предназначен для выполнения операций A, D и Z (см. описание операций над семафорами из массива IPC семафоров – раздел "Создание массива семафоров или доступ к уже существующему.Системный вызов semget()"). Данное описание не является полным описанием системного вызова, а ограничивается рамками текущего курса. Для полного описания обращайтесь к LINUX Manual.

> Параметр semid является дескриптором System V IPC для набора семафоров, т. е. значением, которое вернул системный вызов semget()при создании набора семафоров или при его поиске по ключу.

Каждый из nsops элементов массива, на который указывает параметр sops, определяет операцию, которая должна быть совершена над каким-либо семафором из массива IPC семафоров, и имеет тип структуры structsembuf, в которую входят следующие переменные:

- shortsem_num – номер семафора в массиве IPC семафоров (нумеруются, начиная с 0);
- shortsem_op – выполняемая операция;
- shortsem_flg – флаги для выполнения операции. В нашем курсе всегда будем считать эту переменную равной 0.

Значение элемента структуры sem_op определяется следующим образом:

- для выполнения операции A(S,n) значение должно быть равно n ;
- для выполнения операции D(S,n) значение должно быть равно -n ;
- для выполнения операции Z(S) значение должно быть равно 0.

Семантика системного вызова подразумевает, что все операции будут в реальности выполнены над семафорами только перед успешным возвращением из системного вызова. Если при выполнении операций D или Z процесс перешел в состояние ожидания, то он может быть выведен из этого состояния при возникновении следующих форс-мажорных ситуаций:

- массив семафоров был удален из системы;

- процесс получил сигнал, который должен быть обработан.

В этом случае происходит возврат из системного вызова с констатацией ошибочной ситуации.

Возвращаемое значение

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

2.6 Создание программы с использованием семафора

Для иллюстрации сказанного рассмотрим простейшие программы, синхронизирующие свои действия с помощью семафоров

```
/* Программа 08-1a.c 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 08-1b.c,а затем уменьшает его на 1*/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
int main()
{
        intsemid; /* IPC дескриптордлямассива IPC
семафоров */
        char pathname[] = "08-1a.c"; /* Имяфайла,
                использующееся для генерации ключа. Файл с таким
                именем должен существовать в текущей директории */
        key_t key; /* IPC ключ */
        structsembufmybuf; /* Структура для задания
                операции над семафором */
        /* Генерируем IPC-ключ из имени файла 08-1a.c в текущей
        директории и номера экземпляра массива семафоров 0 */
        if((key = ftok(pathname,0)) < 0){
                printf("Can\'t generate key\n");
                exit(-1);
        }
        /* Пытаемся 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){
```

```
                printf("Can\'t get semid\n");
                exit(-1);
        }
        /* Выполним операцию D(semid1,1) для нашего массива
        семафоров. Для этого сначала заполним нашу структуру.
        Флаг, как обычно, полагаем равным 0. Наш 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){
                printf("Can\'t wait for condition\n");
                exit(-1);
        }
        printf("Condition is present\n");
        return 0;
}
```

Листинг 8.1а. Программа 08-1a.c для иллюстрации работы с семафорами

```
/* Программа 08-1b.c 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>
int main()
{
        intsemid; /* IPC дескриптордлямассива IPC
                семафоров */
        char pathname[] = "08-1a.c"; /* Имяфайла,
                использующееся для генерации ключа. Файл с таким
                именем должен существовать в текущей директории */
        key_t key; /* IPC ключ */
        structsembufmybuf; /* Структура для задания операции
        над семафором */
        /* Генерируем IPC ключ из имени файла 08-1a.c в текущей
        директории и номера экземпляра массива семафоров 0 */
        if((key = ftok(pathname,0)) < 0){
                printf("Can\'t generate key\n");
                exit(-1);
        }
        /* Пытаемся 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){
                printf("Can\'t get semid\n");
                exit(-1);
        }
        /* Выполним операцию A(semid1,1) для нашего массива
        семафоров. Для этого сначала заполним нашу структуру.
        Флаг, как обычно,       полагаем равным 0. Наш 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){
                printf("Can\'t wait for condition\n");
                exit(-1);
        }
        printf("Condition is set\n");
        return 0;
}
```

Листинг 8.1b. Программа 08-1b.c 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 инициируется 0, то программа 1 может работать без блокировки только после запуска программы 2.

2.7 Удаление набора семафоров из системы с помощью команды ipcrm или системного вызова semctl()

Как мы видели в примерах, 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, в этом случае должна иметь вид

ipcrmsem<IPC идентификатор>

Для этой же цели мы можем применять системный вызовsemctl(), который умеет выполнять и другие операции над массивом семафоров, но их рассмотрение выходит за рамки нашего курса.

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,
unionsemunarg);
```

Описание системного вызова

Системный вызов semctlпредназначен для получения информации о массиве IPC семафоров, изменения его атрибутов и удаления его из системы. Данное описание не является полным описанием системного вызова, а ограничивается рамками текущего курса. Для изучения полного описания обращайтесь к LINUX Manual.

В нашем курсе мы будем применять системный вызов semctlтолько 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.

2.8 Понятие о POSIX-семафорах

В стандарте POSIX вводятся другие семафоры, полностью аналогичные семафорам Дейкстры. Для инициализации значения таких семафоров применяется функциsem_init(), аналогом операцииP служит функциsem_wait(), а аналогом операцииV – функциsem_post(). К сожалению, в Linux такие семафоры реализованы только для нитей исполнения одного процесса, и поэтому подробно мы на них останавливаться не будем.

3. Задание к лабораторной работе№8.

1. Ознакомиться с теоретической частью к лабораторной работе.
2. Наберите программы 08-1a.c и 08-1b.c , сохраните их соответственно, откомпилируйте и проверьте правильность их поведения.
3. Измените программы из предыдущего п.2 так, чтобы первая программа могла работать без блокировки после не менее 5 запусков второй программы.
4. (*)В материалах лабораторной работы 7, когда речь шла о связи родственных процессов через pipe, отмечалось, что pipe является однонаправленным каналом связи, и что для организации связи через один pipe в двух направлениях необходимо использовать механизмы взаимной синхронизации процессов. Организуйте двустороннюю поочередную связь процесса-родителя и процесса-ребенка через pipe, используя для синхронизации семафоры, модифицировав программу из раздела "Прогон программы для организации однонаправленной связи между родственными процессами через pipe" лабораторной работы 7.

4. Контрольные вопросы

1. В чем состоит отличие семафорами System V IPC от классического набора операций {P, V}, предложенного Дейкстрой?
2. Каким значением изначально инициируются все IPC-семафоры?
3. Какая операция семафора System V IPC соответствует классической операцииP(S)?
4. Какая операция семафора System V IPC соответствует классической операцииV(S) ?
5. Что может служить аналогом ненулевой инициализации семафоров Дейкстры в семафорах System V IPC?
6. Для чего используется системный вызов semop()?
7. Может ли массив семафоров продолжать существовать в системе и после завершения использовавших его процессов?
8. Для чего применяется системный вызовs emctl()?

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. **Скловская, А.М**. Команды LINUX. Справочник. / А.М. Скловская – Изд-во Диасофт, 2004. – 848 с.
2. **Моли, Б.** Unix/Linux: Теория и практика программирования / Б.Моли – Изд-во КУДИЦ-ОБРАЗ, 2004. – 576 с.
3. **Бендел, Д**. Использование Linux. /Д.Бендел, Р.Нейпир . – М.: издательский дом "Вильямс", 2004. – 784 с
4. **Немет Э**.UNIX: руководство системного администратора. Для профессионалов / Э.Немет, Снайдер Г., Сибас С., Хейн Т.Р. – СП.: Питер; К.: Издательская группа BHV, 2002. – 928 с
5. **Карпов, В.** Основы операционных систем.Практикум [Электронный документ]. (http://www.intuit.ru/studies/courses/2249/52). Проверенно 7.07.2017
6. **Иванов, Н. Н**. Программирование в Linux. Самоучитель. – 2-е изд., перераб. и доп. / Н.Н. Иванов – СПб.: БХВ-Петербург, 2012. – 400 с.

Учебное издание

Москат Наталья Александровна

**ОПЕРАЦИОННЫЕ СИСТЕМЫ
И КРОССПЛАТФОРМЕННОЕ ПРОГРАММИРОВАНИЕ.
ОПЕРАЦИОННАЯ СИСТЕМА LINUX**

Печатается в авторской редакции

Технический редактор А.В. Артамонов

Подписано в печать 04.09.17. Формат 60×84/16.
Бумага газетная. Ризография. Усл. печ. л. 5,81.
Тираж экз. Изд. № 9018. Заказ .

Редакционно-издательский центр ФГБОУ ВО РГУПС.

Адрес университета: 344038, г. Ростов н/Д, пл. Ростовского Стрелкового Полка Народного Ополчения, д. 2.