Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Уфимский государственный авиационный технический университет»

РАБОТА В ОПЕРАЦИОННОЙ СИСТЕМЕ LINUX

Лабораторный практикум
по дисциплине
«Операционные системы»

Часть 1

УФА 2014

Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Уфимский государственный авиационный технический университет»

Кафедра автоматизированных систем управления

РАБОТА В ОПЕРАЦИОННОЙ СИСТЕМЕ LINUX

Лабораторный практикум
по дисциплине
«Операционные системы»

УФА 2014

Составитель: О.Д. Лянцев, А.В. Казанцев

УДК 004.5(07)
ББК 32.973-018.2(я7)

Лабораторный практикум по дисциплине «Операционные системы» / Уфимск. гос. авиац. техн. ун-т; Сост. О.Д. Лянцев, А.В. Казанцев. – Уфа, 2014. – 66 с.

Приведены основные сведения о принципах организации и функционирования многопользовательской операционной системы LINUX. Рассматриваются структура файловой системы, функции по обработке и управлению данными, создание и выполнение командных файлов. Изучаются принципы взаимодействия LINUX с внешними устройствами и принципы средства обмена информацией между процессами. Изучаются многозадачный режим выполнения процессов, а также пользовательский и программный интерфейсы операционной системы. Практическое изучение функциональных особенностей системы иллюстрируется примерами формирования простых и сложных команд по обработке данных. Приводится методика и порядок выполнения лабораторных работ, в приложении содержатся вспомогательные материалы.

Предназначен для подготовки студентов, обучающихся по специальности 230106 – «Применение и эксплуатация автоматизированных систем специального назначения» и 230100 «Информатика и вычислительная техника».

Библиогр.: 6 назв.
Рецензенты: А.А. Колесников, В.Ю. Арьков

СОДЕРЖАНИЕ

Введение

Дисциплина «Операционные системы» предназначена для формирования у студентов систематизированного представления об основополагающих принципах функционирования операционных систем, о концепциях, структурах и механизмах, лежащих в основе работы основных функций современных операционных систем, их характеристик и о современных направлениях их развития.

В лабораторных работах рассматриваются основные принципы функционирования операционной системы LINUX, возможности файловой системы и функций по обработке и управления данными, методы динамического порождения процессов и способы передачи информации между взаимодействующими процессами.

В результате выполнения данного лабораторного практикума формируются следующие компетенции:

– владение знаниями о теоретических основах и основных принципах функционирования, методах программирования пользовательских программ;

– готовность использовать навык применения типовых системных вызовов для создания прикладный программ;

– способность применять систему программирования на языках C и C++.

Лабораторная работа № 1

ВЫПОЛНЕНИЕ КОМАНД В СРЕДЕ ОС LINUX

1. Цель работы

Целью работы является изучение архитектуры и принципов функционирования многопользовательской многозадачной операционной системы Linux, особенности ее использования в качестве рабочей станции.

2. Задачи работы

– Закрепление, углубление и расширение знаний студентов при использовании операционной системы Linux.

– Приобретение умений и навыков работы с командным интерпретатором Bash в операционной системе Linux.

– Выработка способности логического мышления, осмысления полученных результатов при применении системных и встроенных команд интерпретатора Bash.

3. Теоретическая часть

ОС Linux 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 созда-

вать 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.

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

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 (Network File System) 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 (Transfer Control Protocol/Internet Protocol). 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 CC или GCC 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.

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 параметры:

echo good morning

и нажмите клавишу *Enter*. На экране появится приветствие "*good morning*" – параметр команды **echo.** Командный интерпретатор *shell* вызвал команду **echo**, реализованную в виде программы на языке СИ, и передал ей параметры. После этого интерпретатор команд вывел знак-приглашение. Синтаксис команды **echo**:

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

Команда помещает в стандартный вывод свои параметры, 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 параметры 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 **man man** 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 параметр 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** удаляет файлы text.1, text.2, text.3. Другие варианты этой команды - **rm text.[123]** или **rm text.[1-3].**

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

clear - очистка экрана.

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.

who | wc - подсчет количества работающих пользователей командой **wc** (word count - счет слов);

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

mail student < 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;

cc primer.c & - трансляция Си - программы в фоновом режиме. Имя выполняемой программы по умолчанию a.out.

cc -o primer.o primer.c - трансляция Си-программы с образованием файла выполняемой программы с именем primer.o;

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 ак-

тивных 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) *Midnight Commander*, аналогичного *Norton Commander*.

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

4. Краткое описание командного интерпретатора Shell

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.

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

5. Задание на лабораторную работу

С помощью командного интерпретатора Shell набрать и выполнить команды, исходный текст которых приведен в примерах.

6. Методика выполнения задания

Порядок выполнения работы:

1. Ознакомиться с теоретической частью к лабораторной работе.
2. Определить день недели, в который Вы родились.
3. Получить подробную информацию обо всех активных процессах.

4. Используя редактор VI (см. приложение), создать два текстовых файла (с расширением TXT) и командой CAT просмотреть их на экране.

5. Получить информацию о работающих пользователях, подсчитать их количество и запомнить в файле.

6. Объединить текстовые файлы в единый файл и посмотреть его на экране.

7. Посмотреть приоритет своего процесса и уменьшить скорость его выполнение за счет повышения номера приоритета.

8. Используя редактор VI, написать программу на языке Си и запустить ее на трансляцию в фоновом режиме.

9. Показать преподавателю исходный текст программы на языке Си, текстовый файл, файл с сохранением количества пользователей.

10. Продемонстрировать выполнение Си - программы.

11. Удалить свои файлы и выйти из системы.

7. Требования к содержанию и оформлению отчета

Отчет по лабораторной работе должен содержать:

а) титульный лист;

б) исходный текст выполненных программ;

в) результаты, выведенные программами на экран дисплея;

г) ответы на контрольные вопросы.

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

ФУНКЦИИ ФАЙЛОВОЙ СИСТЕМЫ 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. Задачи работы

– Закрепление, углубление и расширение знаний студентов при использовании файловой системы ОС Linux.

– Приобретение умений и навыков работы с функциями файловой системы по модификации файлов и каталогов в операционной системе Linux.

– Выработка способности логического мышления, осмысления полученных результатов при применении файловых команд ОС Linux.

3. Теоретическая часть

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          rwx          rwx          rwx
|           |            |            |
|           |            |            | 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 - перемещение файлов file.1, file.2, file.3 в указанную директорию;

rm file.1 file.2 file.3 - удаление файлов file.1, file.2, file.3;.

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

mkdir namedir - 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 -l student *.txt

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

who | grep tty23

4. Краткое описание командного интерпретатора Shell

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.

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

5. Задание на лабораторную работу

С помощью командного интерпретатора Shell набрать и выполнить команды, исходный текст которых приведен в примерах.

6. Методика выполнения задания

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.

7. Требования к содержанию и оформлению отчета

Отчет по лабораторной работе должен содержать:

а) титульный лист;

б) исходный текст выполненных программ;

в) результаты, выведенные программами на экран дисплея;

г) ответы на контрольные вопросы.

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 № 3

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. Задачи работы

– Закрепление, углубление и расширение знаний студентов при создании и выполнения командных скриптов в ОС Linux.

– Приобретение умений и навыков работы с shell-файлами в операционной системе Linux.

– Выработка способности логического мышления, осмысления полученных результатов при применении командных сценариев в ОС Linux.

3. 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.

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 интерпре-

татором 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.

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*.

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 cmd выполняется 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;

read name - 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:"
read name
```

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.

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 [-arwfds] file

где

-a – файл существует;

-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 **if test true** эквивалентно выражению **if test ! 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.

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. Формат оператора цикла с условием **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 соот-

ветствующими 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.

5. Задание на лабораторную работу

С помощью командного интерпретатора Shell набрать и выполнить команды, исходный текст которых приведен в примерах.

6. Методика выполнения задания

Составьте и выполните shell - программы, включающей следующие действия:

1. Создать каталог DIR и в нем создать файл Myfile.txt, записать в файл свою фамилию и текущее время.

2. Переименовать файл Myfile.txt в Old_Myfile.txt и установить у него атрибут ReadOnly.

3. Вычислить омическое сопротивление двух параллельно соединенных резисторов. Значения сопротивлений резисторов в диапазоне от 0,1 ом до 1000 Мом вводить с клавиатуры.

4. Перевести заданное десятичное число в шестнадцатеричную форму. Ввод десятичного числа с клавиатуры в диапазоне от 0 до 4096.

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.

7. Требования к содержанию и оформлению отчета

Отчет по лабораторной работе должен содержать:

а) титульный лист;

б) исходный текст выполненных программ;

в) результаты, выведенные программами на экран дисплея;

г) ответы на контрольные вопросы.

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 №4
Функции для создания процессов

1. Цель работы

Целью работы является изучение методов программирования по созданию пользовательских процессов в ОС Linux.

2. Задачи работы

– Закрепление, углубление и расширение знаний студентов при использовании процессов в прикладных программах.

– Приобретение умений и навыков работы с системой программирования на языках C и C++ в операционной системе Linux.

– Выработка способности логического мышления, осмысления полученных результатов при применении набора функций по созданию процессов.

3. Теоретическая часть

Идентификаторы процессов. Каждый процесс в Linux помечается уникальным идентификатором (PID , process identifer). Идентификаторы – это 16-разрядные числа, назначаемые последовательно по мере создания процессов. У всякого процесса имеется также родительский процесс за исключением специального суперсервера init с идентификатором 1. Таким образом, все процессы Linux организованы в виде сложной иерархии, на вершине которой находится процесс **init**. Иерархию процессов можно увидеть, выполнив команду **ps –axf** . К атрибутам процесса относится идентификатор его родительского процесса (PPID, parent process identifer).

Работая с идентификаторами процессов в программах, написанных на языках C и C++, следует объявить соответствующие переменные как имеющие тип **pid_t** (этот тип определяется в файле **<sys/types.h>)**. Программа может узнать идентификатор своего собственного процесса с помощью системного вызова **getpid()**, а идентификатор своего родительского процесса с помощью системного вызова **getppid()**.

Пример 1:

```
#include<stdio.h>
#include<unistd.h>
int main()
```

```
{
printf (“Номер процесса: %d\n”, (int) getpid() );
printf («Номер родительского процесса: %d\n», (int) getppid());
return 0;
}
```

Обратите внимание на важную особенность: при каждом вызове программа сообщает о разных идентификаторах, поскольку всякий раз запускается новый процесс. Тем не менее, если программа вызывается из одного и того же интерпретатора команд, то родительский идентификатор оказывается одинаковым.

Создание процессов

Существуют два способа создания процессов. Первый из них относительно прост, применяется редко, поскольку неэффективен и связан со значительным риском для безопасности системы. Второй способ сложнее, но избавлен от недостатков первого. Первый способ основан на применении функции **system()**, а второй на основе применения функций **fork()** и **exec()**.

Функция system()

Функция **system()** определена в стандартной библиотеке языка С и позволяет вызвать из программы системную команду, как если бы она была набрана в командной строке.

По сути, эта функция запускает стандартный интерпретатор и передает ему команду на выполнение.

Пример 2:

```
#include<stdlib.h>
int main()
{
int return_value;
return_value = system(“ls –l /”);
return return_value;
}
```

Функция **system()** возвращает код завершения указной команды. Если интерпретатор не может быть запущен, возвращается значения 127, а в случае возникновения других ошибок (-1).

Поскольку функция **system()** запускает интерпретатор команд, она подвержена всем тем ограничениям безопасности, что и командный интерпретатор.

В большинстве Unix-системах программа /bin/sh представляет собой символическую ссылку на другой интерпретатор. В Linux это в основном bash.

Вызов из функции **system()** программы с привилегиями пользователя root также может иметь неодинаковые последствия в разных системах. Таким образом, лучше создавать процессы с помощью функций **fork()** и **exec()**.

Функции fork() и exec()

В DOS и Windows API имеется семейство функций **spawn()**. Они принимают в качестве аргумента имя программы, создают новый экземпляр ее процесса и запускают его.

В Linux нет такой функции, которая выполнила бы все это за одни заход. Вместо этого имеются функция **fork()**, создающая дочерний процесс, который является точной копией родительского процесса, и семейство функций **exec()**, заставляющих требуемый процесс перестать быть вторым экземпляром одной программы и превратиться в экземпляр другой программы.

Чтобы создать новый процесс, нужно сначала с помощью функции **fork()** создать копию текущего процесса, а затем с помощью функции **exec()** преобразовать одну из копий в экземпляр запускаемой программы.

Вызов функции fork()

Вызывая функцию **fork()**, программа создает свой дубликат, называемый дочерним процессом. Родительский процесс продолжает выполнять программу с той точки, где была вызвана функция **fork().** То же самое делает и дочерний процесс.

Процессы отличаются своими идентификаторами. Таким образом, программа может вызывать функцию **getpid()** и узнать где именно она находится.

Но сама функция **fork()** реализует другой способ: она возвращает разные значения в родительском и дочернем процессах. В родительском процессе функция **fork()** равна идентификатору своего потомка, а в дочернем процессе она равна 0. Рассмотрим данную ситуацию на примере, учтите, что первая часть инструкции **if** выполняется только в родительском процессе, тогда как ветвь **else** – только в дочернем.

Пример 3:

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main()
{
pid_t child_pid;
printf(“ID процесса основной программы: %d\n”, (int) getpid() );
child_pid = fork();
if (child_pid)
{
printf(«Это родительский процесс, с ID %d\n», (int) getpid() );
printf(“Дочерний процесс, с ID %d\n”, (int) child_pid );
}
else
printf(«Дочерний процесс с ID %d\n», (int) getpid() );
return 0;
}
```

Семейство функций exec()

Функции семейства **exec()** заменяют программу, выполняющуюся в текущем процессе, другой программой. Когда программа вызывает функцию **exec()**, ее выполнение немедленно прекращается и начинает работу новая программа.

Функции, в название которых присутствует суффикс 'p' (**execvp()** и **execlp()**), принимают в качестве аргумента имя программы и ищут эту программу в каталогах, определяемых переменной среды PATH. Всем остальным функциям нужно передавать полное путевое имя программы.

Функции, в названии которых присутствует суффикс 'v' (**execv()**, **execvp()**, **execve()**), принимают список аргументов программы в виде массива строковых указателей, оканчивающегося NULL-указателем. Функции с суффиксом 'l' (**excevl()**, **execlp()**, **execlve()**), принимают список аргументов переменного размера.

Функции, в названии которых присутствует суффикс 'e' (**execve()**, **execle()**), в качестве дополнительного аргумента принимают массив переменных среды. Этот массив содержит строковые указатели и оканчивается пустым указателем. Каждая строка должна иметь вид «Переменная = значение».

Поскольку функция **exec()** заменяет одну программу другой, она никогда не возвращает значение – только если вызов программы оказался невозможен в случае ошибки.

Список аргументов, передаваемых программе, аналогичен аргументам командной строки, указываемым при запуске программы в интерактивном режиме. Их тоже можно получить с помощью параметров **argc** и **argv** функции **main()**. Когда программу запускает интерпретатор команд, первый элемент массива **argv** будет содержать имя программы, а далее будут находиться переданные программе аргументы. Аналогичным образом следует поступить, формируя список аргументов для функции **exec()**.

Совместное использование функций fork() и exec()

Стандартная методика запуска одной программы из другой такова: сначала с помощью функции **fork()** создается дочерний процесс, затем в нем вызывается функция **exec()**.

Это позволяет главной программе продолжать выполнение в родительском процессе. В качестве примера напишем программу, которая отображает корневой каталог.

Пример 4:

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<unistd.h>
int spawn(char* program, char** arg_list)
{
pid_t child_pid;
child_pid = fork();
if(child_pid)
return child_pid;
else
{
execvp (program, arg_list);
fprintf (stderr, “an error    роцесс    in execvp\n”);
abort();
}
}
int main()
{
int child_status;
```

```
char* arg_list[] = {“ls”,”-l”,”/”,NULL};
spawn (“ls”, arg_list);
wait (&child_status);
printf(“done\n”);
return 0;
}
```

Системные вызовы wait()

Самая простая функция в семействе называется **wait()**. Она блокирует вызывающий процесс до тех пор, пока один из его дочерних процессов не завершиться (или не произойдет ошибка).

Пример использования данной функции приведен выше.

Функция **waitpid()** позволяет дождаться завершения конкретного дочернего процесса.

Функция **wait3()** возвращает информацию о статистике использования центрального процессора завершившемся дочерним роцесссом.

Функция **wait4()** позволяет задать дополнительную информацию о том, каких процессов следует дождаться.

4. Краткое описание языка программирования Си

Си – универсальный язык программирования. Он тесно связан с системой UNIX, так как был разработан в этой системе, которая, как и большинство программ работающих в ней, написаны на Си.

В отличие от «безтиповых» языков Си обеспечивает разнообразие типов данных. Базовыми типами являются символы, а также целые и числа с плавающей точкой различных размеров. Кроме того, имеется возможность получать целую иерархию производных типов данных из указателей, массивов, структур и объединений. Выражения формируются из операторов и операндов. Любое выражение, включая присваивание и вызов функции, может быть инструкцией. Указатели обеспечивают машинно-независимую адресную арифметику. В Си имеются основные управляющие конструкции, используемые в хорошо структурированных программах: составная инструкция ({...}), ветвление по условию (if-else), выбор одной альтернативы из многих (switch), циклы с проверкой наверху (while, for) и с проверкой внизу (do), а также средство прерывания цикла (break). В качестве результата функции могут возвращать значения базовых типов, структур, объединений и указателей. Любая функция допускает рекурсивное

обращение к себе. Функции программы на Си могут храниться в отдельных исходных файлах и компилироваться независимо. Переменные по отношению к функции могут быть внутренними и внешними. Последние могут быть доступными в пределах одного исходного файла или всей программы. Си – язык сравнительно «низкого уровня». Однако это вовсе не умаляет его достоинств, просто Си имеет дело с теми же объектами, что и большинство компьютеров, т. е. с символами, числами и адресами. С ними, можно оперировать при помощи арифметических и логических операций, выполняемых реальными машинами. В Си нет прямых операций над составными объектами, такими как строки символов, множества, списки и массивы. В нем нет операций, которые бы манипулировали с целыми массивами или строками символов, хотя структуры разрешается копировать целиком как единые объекты. В языке нет каких-либо средств распределения памяти, помимо возможности определения статических переменных и стекового механизма при выделении места для локальных переменных внутри функций. Наконец, в самом Си нет средств ввода-вывода, инструкций READ (читать) и WRITE (писать) и каких-либо методов доступа к файлам. Все это – механизмы высокого уровня, которые в Си обеспечиваются исключительно с помощью явно вызываемых функций. Большинство реализованных Си-систем содержат в себе разумный стандартный набор этих функций. Си предоставляет средства лишь последовательного управления ходом вычислений: механизм ветвления по условиям, циклы, составные инструкции, подпрограммы – и не содержит средств мультипрограммирования, параллельных процессов, синхронизации и организации сопрограмм. Однако компактность языка имеет реальные выгоды. Поскольку Си относительно мал, то и описание его кратко, и овладеть им можно быстро. Программист может реально рассчитывать на то, что он будет знать, понимать и на практике регулярно пользоваться всеми возможностями языка. Важный аспект языка – это определение библиотеки, поставляемой вместе с Си-компилятором, в которой специфицируются функции доступа к возможностям операционной системы (например, чтения-записи файлов), форматного ввода-вывода, динамического выделения памяти, манипуляций со строками символов и т. д. Набор стандартных заголовочных файлов обеспечивает единообразный доступ к объявлениям функций и типов данных. Почти все программы, написанные на Си, если они не касаются каких-

либо скрытых в операционной системе деталей, переносимы на другие машины. Си соответствует аппаратным возможностям многих машин, однако он не привязан к архитектуре какой-либо конкретной машины. Основной философией Си остается то, что программисты сами знают, что делают; язык лишь требует явного указания об их намерениях. Си, как и любой другой язык программирования, не свободен от недостатков. Тем не менее, как оказалось, Си – чрезвычайно эффективный и выразительный язык, пригодный для широкого класса задач.

5. Задание на лабораторную работу

С помощью компилятора С создать и выполнить программы, исходный текст которых приведен в примерах 1 – 4.

6. Методика выполнения задания

Порядок выполнения работы:

1. Прочитать методический материал.
2. Изучить характеристики и синтаксис функций и системных вызовов.
3. Набрать код примеров в текстовые файлы и произвести компиляцию программ.
4. Проверить работоспособность программ.

7. Требования к содержанию и оформлению отчета

Отчет по лабораторной работе должен содержать:

а) титульный лист;

б) исходный текст выполненных программ;

в) результаты, выведенные программами на экран дисплея;

г) ответы на контрольные вопросы.

Контрольные вопросы

1. Что является атрибутами процесса?
2. Как организуется взаимодействие процессов?
3. Каким образом программные средства Linux позволяют динамически порождать процессы?
4. Какие существуют формы системного вызова exec()?

Лабораторная работа №5
Потоковая передача информации между процессом и файлом

1. Цель работы

Целью работы является изучение методов программирования по работе с файлами через системные вызовы и стандартную библиотеку ввода-вывода для языка С.

2. Задачи работы

– Закрепление, углубление и расширение знаний студентов при использовании процессов и файлов в прикладных программах.

– Приобретение умений и навыков работы с системой программирования на языках С и С++ в операционной системе Linux.

– Выработка способности логического мышления, осмысления полученных результатов при применении набора функций для работы с файлами через системные вызовы и стандартную библиотеку ввода-вывода для языка С.

3. Теоретическая часть.

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.

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.

Функции работы с файлами из стандартной библиотеки ввода-вывода, такие как *fopen(), fread(), fwrite(), fprintf(), fscanf(), fgets()* и т.д. входят как неотъемлемая часть в стандарт ANSI на язык С. Они 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.

Файловый дескриптор. 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.

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 дан-

ных из файла и записи их в файл, необходимо поместить информацию о файле в таблицу открытых файлов и определить соответствующий файловый дескриптор. Для этого применяется процедура открытия файла, осуществляемая системным вызовом *open()*.

Прототип системного вызова:

```
#include <fcntl.h>
int open(char *path, int flags);
int open(char *path, int flags, int mode);
```

Системный вызов *open()* предназначен для выполнения операции открытия файла и, в случае ее удачного осуществления, возвращает файловый дескриптор открытого файла (небольшое неотрицательное целое число, которое используется в дальнейшем для других операций с этим файлом). Параметр *path* является указателем на строку, содержащую полное или относительное имя файла. Параметр *flags* может принимать одно из следующих трех значений:

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

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

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

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

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

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

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

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

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

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

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

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

Параметр *mode* устанавливает атрибуты прав доступа различных категорий пользователей к новому файлу при его создании. Он обязателен, если среди заданных флагов присутствует флаг O_CREAT, и может быть опущен в противном случае. Этот параметр задается как сумма следующих восьмеричных значений:

0400 – разрешено чтение для пользователя, создавшего файл;
0200 – разрешена запись для пользователя, создавшего файл;
0100 – разрешено исполнение для пользователя, создавшего файл;
0040 – разрешено чтение для группы пользователя, создавшего файл;
0020 – разрешена запись для группы пользователя, создавшего файл;
0010 – разрешено исполнение для группы пользователя, создавшего файл;
0004 – разрешено чтение для всех остальных пользователей;
0002 – разрешена запись для всех остальных пользователей;
0001 – разрешено исполнение для всех остальных пользователей.

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

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

Системный вызов *open()* 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 лабораторной работе понадобятся 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 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.

Если файл на диске отсутствует и его нужно создать, 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.

Чтобы создать файл в момент открытия, флаг для набора операций должен использоваться в комбинации с флагами O_CREAT и O_EXCL.

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

Прототипы системных вызовов

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

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

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

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

Параметр *nbytes* для системного вызова *write* определяет количество байт, которое должно быть передано, начиная с адреса памяти *addr*. Параметр *nbytes* для системного вызова *read* определяет количество байт, которое необходимо получить из канала связи и разместить в памяти, начиная с адреса *addr*.

В случае успешного завершения системный вызов возвращает количество реально посланных или принятых байт. 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.

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.

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()*. 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(int fd);
```

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

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

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

В качестве иллюстрации рассмотрим следующую программу:

```
/*Программа 05
-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(){
   int fd;
   size_t size;
   char string[] = "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 и
       прекращаем работу */
      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);
    }
    /* Закрываем файл */
    if(close(fd) < 0){
        printf("Can\'t close file\n");
    }
    return 0;
}
```

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()*.

4. Краткое описание языка программирования Си

Си – универсальный язык программирования. Он тесно связан с системой UNIX, так как был разработан в этой системе, которая, как и большинство программ работающих в ней, написаны на Си.

В отличие от «безтиповых» языков Си обеспечивает разнообразие типов данных. Базовыми типами являются символы, а также целые и числа с плавающей точкой различных размеров. Кроме того, имеется возможность получать целую иерархию производных типов данных из указателей, массивов, структур и объединений. Выражения формируются из операторов и операндов. Любое выражение, включая присваивание и вызов функции, может быть инструкцией. Указатели обеспечивают машинно-независимую адресную арифметику. В Си имеются основные управляющие конструкции, используемые в хорошо структурированных программах: составная инструкция ({...}), ветвление по условию (if-else), выбор одной альтернативы из многих (switch), циклы с проверкой наверху (while, for) и с проверкой внизу (do), а также средство прерывания цикла (break). В качестве результата функции могут возвращать значения базовых типов, структур, объединений и указателей. Любая функция допускает рекурсивное обращение к себе. Функции программы на Си могут храниться в отдельных исходных файлах и компилироваться независимо. Переменные по отношению к функции могут быть внутренними и внешними. Последние могут быть доступными в пределах одного исходного файла или всей программы. Си – язык сравнительно «низкого уров-

ня». Однако это вовсе не умаляет его достоинств, просто Си имеет дело с теми же объектами, что и большинство компьютеров, т. е. с символами, числами и адресами. С ними, можно оперировать при помощи арифметических и логических операций, выполняемых реальными машинами. В Си нет прямых операций над составными объектами, такими как строки символов, множества, списки и массивы. В нем нет операций, которые бы манипулировали с целыми массивами или строками символов, хотя структуры разрешается копировать целиком как единые объекты. В языке нет каких-либо средств распределения памяти, помимо возможности определения статических переменных и стекового механизма при выделении места для локальных переменных внутри функций. Наконец, в самом Си нет средств ввода-вывода, инструкций READ (читать) и WRITE (писать) и каких-либо методов доступа к файлам. Все это – механизмы высокого уровня, которые в Си обеспечиваются исключительно с помощью явно вызываемых функций. Большинство реализованных Си-систем содержат в себе разумный стандартный набор этих функций. Си предоставляет средства лишь последовательного управления ходом вычислений: механизм ветвления по условиям, циклы, составные инструкции, подпрограммы – и не содержит средств мультипрограммирования, параллельных процессов, синхронизации и организации сопрограмм. Однако компактность языка имеет реальные выгоды. Поскольку Си относительно мал, то и описание его кратко, и овладеть им можно быстро. Программист может реально рассчитывать на то, что он будет знать, понимать и на практике регулярно пользоваться всеми возможностями языка. Важный аспект языка – это определение библиотеки, поставляемой вместе с Си-компилятором, в которой специфицируются функции доступа к возможностям операционной системы (например, чтения-записи файлов), форматного ввода-вывода, динамического выделения памяти, манипуляций со строками символов и т. д. Набор стандартных заголовочных файлов обеспечивает единообразный доступ к объявлениям функций и типов данных. Почти все программы, написанные на Си, если они не касаются каких-либо скрытых в операционной системе деталей, переносимы на другие машины. Си соответствует аппаратным возможностям многих машин, однако он не привязан к архитектуре какой-либо конкретной машины. Основной философией Си остается то, что программисты сами знают, что делают; язык лишь требует явного указания об их

намерениях. Си, как и любой другой язык программирования, не свободен от недостатков. Тем не менее, как оказалось, Си – чрезвычайно эффективный и выразительный язык, пригодный для широкого класса задач.

5. Задание на лабораторную работу

С помощью компилятора С создать и выполнить программу, исходный текст которой приведен в примере 1.

Модифицировать эту программу для вывода на экран информации о результате создания файла.

6. Методика выполнения задания

Порядок выполнения работы:

1. Прочитать методический материал.
2. Изучить характеристики и синтаксис функций и системных вызовов.
3. Набрать код примера 1 в текстовый файл и произвести компиляцию программы.
4. Проверить работоспособность программы.
5. Модифицировать программу для вывода на экран сообщение о результате создания файла.
6. Проверить работоспособность модифицированной программы.

7. Требования к содержанию и оформлению отчета

Отчет по лабораторной работе должен содержать:

а) титульный лист;

б) исходный текст выполненных программ;

в) результаты, выведенные программой на экран дисплея;

г) ответы на контрольные вопросы.

Контрольные вопросы

1. Какие системные вызовы используются для записи информации в файл?
2. Как формируются права доступа к файлу, создаваемому пользовательской программой?
3. Для чего предназначен системный вызов *close*?
4. По каким признакам можно определить неудачную попытку открыть файл?

5. Чем отличаются системные вызовы *fread()* и *read()*?

Лабораторная работа № 6
Передача информации между процессами. Системный вызов pipe

1. Цель работы

Целью работы является изучение методов программирования по взаимодействию процессов через системные вызовы и стандартную библиотеку ввода-вывода.

2. Задачи работы

– Закрепление, углубление и расширение знаний студентов при использовании процессов и файлов в прикладных программах.

– Приобретение умений и навыков работы с системой программирования на языках C и C++ в операционной системе Linux.

– Выработка способности логического мышления, осмысления полученных результатов при применении набора функций для работы с процессами через системные вызовы и стандартную библиотеку ввода-вывода для языка C.

3. Теоретическая часть

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 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. 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 вызов *pipe ()*.

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. 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 используются 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. 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.

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 06-1.c, иллюстрирующая работу с pip'ом в рамках одного
```

```
процесса */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include<stdlib.h>
int main(){
   int fd[2];
   size_t size;
   char string[] = "Hello, world!";
   char resstring[14];
   /* Попытаемся создать pipe */
   if(pipe(fd) < 0)
{
      /* Если создать pipe не удалось, печатаем об этом сообщение
      и прекращаем работу */
      printf("Can\'t create pipe\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
   записанную строку */
   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);
   /* Закрываем входной поток*/
   if(close(fd[0]) < 0){
      printf("Can\'t close input stream\n");
   }
   /* Закрываем выходной поток*/
   if(close(fd[1]) < 0){
```

```
        printf("Can\'t close output stream\n");
    }
    return 0;
}
```

Организация связи через *pipe* между процессом-родителем и процессом-потомком

Достоинство *pip'ов* не сводится к замене функции копирования из памяти в память внутри одного процесса для пересылки информации через операционную систему. 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()*. Это обстоятельство позволяет организовать передачу информации через *pipe* между родственными процессами, имеющими общего прародителя, создавшего *pipe*.

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

```
/* Программа 06-2.c, осуществляющая однонаправленную связь через pipe между процессом-родителем и процессом-ребенком */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include<stdlib.h>
int main(){
    int fd[2], result;
    size_t size;
    char resstring[14];
    /* Попытаемся создать pipe */
    if(pipe(fd) < 0){
        /* Если создать pipe не удалось, печатаем об этом сообщение
        и прекращаем работу */
        printf("Can\'t create pipe\n");
        exit(-1);
    }
    /* Порождаем новый процесс */
    result = fork();
    if(result < 0){
        /* Если создать процесс не удалось, сообщаем об этом и
```

```
    завершаем работу */
    printf("Can\'t fork child\n");
    exit(-1);
} else if (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 pipe, может их использовать.
    В этом процессе входной поток данных нам не
    понадобится, поэтому закрываем его.*/
    close(fd[1]);
    /* Пробуем прочитать из pip'a 14 байт в массив, т.е. всю
    записанную строку */
    size = read(fd[0], resstring, 14);
    if(size < 0){
        /* Если прочитать не смогли, сообщаем об ошибке и
        завершаем работу */
        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;
}
```

Организации двунаправленной связи между родственными процессами через pipe.

Pipe служит для организации однонаправленной или симплексной связи. Если бы в предыдущем примере попытаться организовать через *pipe* двустороннюю связь, когда процесс-родитель пишет информацию в *pipe* , предполагая, что ее получит процесс-ребенок, а затем читает информацию из *pip'a*, предполагая, что ее записал порожденный процесс, то могла бы возникнуть ситуация, в которой процесс-родитель прочитал бы собственную информацию, а процесс-ребенок не получил бы ничего. Для использования одного *pip'a* в двух направлениях необходимы специальные средства синхронизации процессов. Более простой способ организации двунаправленной связи между родственными процессами заключается в использовании двух *pipe*. Модифицируйте программу из предыдущего примера для организации такой двусторонней связи, откомпилируйте ее и запустите на исполнение.

Необходимо отметить, что в некоторых UNIX-подобных системах (например, в Solaris2) реализованы полностью дуплексные *pip'ы*. В таких системах для обоих файловых дескрипторов, ассоциированных с *pip'ом*, разрешены и операция чтения, и операция записи. Однако такое поведение не характерно для *pip'ов* и не является переносимым.

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

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

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!

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

Попытка прочитать меньше байт, чем есть в наличии в канале связи приводит к чтению требуемого количества байт. При этом возвращается значение, соответствующее прочитанному количеству. Прочитанная информация удаляется из канала связи.

В канале связи находится меньше байт, чем затребовано, но не нулевое количество. Читает все, что есть в канале связи, и возвращает значение, соответствующее прочитанному количеству. Прочитанная информация удаляется из канала связи.

Попытка читать из канала связи, в котором нет информации. Блокировка вызова разрешена. Вызов блокируется до тех пор, пока не появится информация в канале связи и пока существует процесс, который может передать в него информацию. Если информация появилась, то процесс разблокируется, и поведение вызова определяется двумя предыдущими строками таблицы. Если в канал некому передать данные (нет ни одного процесса, у которого этот канал связи открыт для записи), то вызов возвращает значение 0. Если канал связи полностью закрывается для записи во время блокировки читающего процесса, то процесс разблокируется, и системный вызов возвращает значение 0.

Попытка читать из канала связи, в котором нет информации. Блокировка вызова не разрешена. Если есть процессы, у которых канал связи открыт для записи, системный вызов возвращает значение -1 и устанавливает переменную *errno* в значение EAGAIN. Если таких процессов нет, системный вызов возвращает значение 0.

Попытка записать в канал связи меньше байт, чем осталось до его заполнения. Требуемое количество байт помещается в канал связи, возвращается записанное количество байт.

Попытка записать в канал связи больше байт, чем осталось до его заполнения. Блокировка вызова разрешена. Вызов блокируется до тех пор, пока все данные не будут помещены в канал связи. Если размер буфера канала связи меньше, чем передаваемое количество информации, то вызов тем самым будет ждать, пока часть информации не будет считана из канала связи. Возвращается записанное количество байт.

Попытка записать в канал связи больше байт, чем осталось до его заполнения, но меньше, чем размер буфера канала связи. Блокировка вызова запрещена. Системный вызов возвращает значение -1 и устанавливает переменную *errno* в значение EAGAIN.

В канале связи есть место. Попытка записать в канал связи больше байт, чем осталось до его заполнения, и больше, чем размер буфера канала связи. Блокировка вызова запрещена. Записывается столько байт, сколько осталось до заполнения канала. Системный вызов возвращает количество записанных байт.

Попытка записи в канал связи, в котором нет места. Блокировка вызова не разрешена. Системный вызов возвращает значение -1 и устанавливает переменную *errno* в значение EAGAIN.

Попытка записи в канал связи, из которого некому больше читать, или полное закрытие канала на чтение во время блокировки системного вызова. Если вызов был заблокирован, то он разблокируется. Процесс получает сигнал SIGPIPE. Если этот сигнал обрабатывается пользователем, то системный вызов вернет значение -1 и установит переменную *errno* в значение EPIPE.

Необходимо отметить дополнительную особенность системного вызова *write* при работе с *pip'ами* и *FIFO*. Запись информации, размер которой не превышает размер буфера, должна осуществляться атомарно – одним подряд лежащим куском. Этим объясняется ряд блокировок и ошибок в предыдущем перечне.

4. Краткое описание языка программирования Си

Си – универсальный язык программирования. Он тесно связан с системой UNIX, так как был разработан в этой системе, которая, как и большинство программ работающих в ней, написаны на Си.

В отличие от «безтиповых» языков Си обеспечивает разнообразие типов данных. Базовыми типами являются символы, а также целые и числа с плавающей точкой различных размеров. Кроме того, имеется возможность получать целую иерархию производных типов данных из указателей, массивов, структур и объединений. Выражения формируются из операторов и операндов. Любое выражение, включая присваивание и вызов функции, может быть инструкцией. Указатели обеспечивают машинно-независимую адресную арифметику. В Си имеются основные управляющие конструкции, используемые в хорошо структурированных программах: составная инструкция ({...}), ветвление по условию (if-else), выбор одной альтернативы из многих (switch), циклы с проверкой наверху (while, for) и с проверкой внизу (do), а также средство прерывания цикла (break). В качестве результата функции могут возвращать значения базовых типов, структур, объединений и указателей. Любая функция допускает рекурсивное обращение к себе. Функции программы на Си могут храниться в отдельных исходных файлах и компилироваться независимо. Переменные по отношению к функции могут быть внутренними и внешними. Последние могут быть доступными в пределах одного исходного файла или всей программы. Си – язык сравнительно «низкого уровня». Однако это вовсе не умаляет его достоинств, просто Си имеет дело с теми же объектами, что и большинство компьютеров, т. е. с символами, числами и адресами. С ними можно оперировать при помощи арифметических и логических операций, выполняемых реальными машинами. В Си нет прямых операций над составными объектами, такими как строки символов, множества, списки и массивы. В нем нет операций, которые бы манипулировали с целыми массивами или строками символов, хотя структуры разрешается копировать целиком как единые объекты. В языке нет каких-либо средств распределения памяти, помимо возможности определения статических переменных и стекового механизма при выделении места для локальных переменных внутри функций. Наконец, в самом Си нет средств ввода-вывода, инструкций READ (читать) и WRITE (писать) и каких-либо методов доступа к файлам. Все это – механизмы высокого уровня, которые в Си обеспечиваются исключительно с помощью явно вызываемых функций. Большинство реализованных Си-систем содержат в себе разумный стандартный набор этих функций. Си предоставляет средства лишь последовательного управления ходом вычис-

лений: механизм ветвления по условиям, циклы, составные инструкции, подпрограммы – и не содержит средств мультипрограммирования, параллельных процессов, синхронизации и организации сопрограмм. Однако компактность языка имеет реальные выгоды. Поскольку Си относительно мал, то и описание его кратко, и овладеть им можно быстро. Программист может реально рассчитывать на то, что он будет знать, понимать и на практике регулярно пользоваться всеми возможностями языка. Важный аспект языка – это определение библиотеки, поставляемой вместе с Си-компилятором, в которой специфицируются функции доступа к возможностям операционной системы (например, чтения-записи файлов), форматного ввода-вывода, динамического выделения памяти, манипуляций со строками символов и т. д. Набор стандартных заголовочных файлов обеспечивает единообразный доступ к объявлениям функций и типов данных. Почти все программы, написанные на Си, если они не касаются каких-либо скрытых в операционной системе деталей, переносимы на другие машины. Си соответствует аппаратным возможностям многих машин, однако он не привязан к архитектуре какой-либо конкретной машины. Основной философией Си остается то, что программисты сами знают, что делают; язык лишь требует явного указания об их намерениях. Си, как и любой другой язык программирования, не свободен от недостатков. Тем не менее, как оказалось, Си – чрезвычайно эффективный и выразительный язык, пригодный для широкого класса задач.

5. Задание на лабораторную работу

С помощью компилятора Си создать и выполнить программы, исходный текст которых приведен в примерах.

6. Методика выполнения задания

Порядок выполнения работы:

1. Прочитать методический материал.

2. Изучить характеристики и синтаксис функций и системных вызовов.

3. Набрать код примеров в текстовые файлы и произвести компиляцию программ.

4. Проверить работоспособность программ.

7. Требования к содержанию и оформлению отчета

Отчет по лабораторной работе должен содержать:

а) титульный лист;

б) исходный текст выполненных программ;

в) результаты, выведенные программой на экран дисплея;

г) ответы на контрольные вопросы.

Контрольные вопросы

1. Что представляет собой в действительности канал *pipe*?
2. Почему в Linux не реализованы полностью дуплексные *pip'ы*?
3. В чем заключается отличие *pip'a* от файла?
4. Что происходит в системе при работе системного вызова *pipe*?
5. Какую информацию содержит переменная *errno*?

Список литературы

1. Скловская А.М. Команды LINUX. Справочник. Изд-во Диасофт. 2012. – 848 с.

2. Моли Б. Unix/Linux: Теория и практика программирования. Изд-во КУДИЦ-ОБРАЗ, 2010. – 576 с.

3. Бендел Д., Нейпир Р. Использование Linux. /Пер.с англ. - М.: издательский дом "Вильямс", 2006. - 784 с.

4. Немет Э., Снайдер Г., Сибас С., Хейн Т.Р. UNIX: руководство системного администратора. Для профессионалов / Пер. с англ. – СП.: Питер; К.: Издательская группа BHV, 2012. – 928 с.

5. http://www.linuxjournal.com.

6. http://pluto.xTech.RU/Russian/Unix-Doc/ - сервер Новосибирского института систем информатики. Содержит книги и документацию по UNIX на русском языке.

Приложение

ПРИНЦИПЫ РАБОТЫ И ОСНОВНЫЕ КОМАНДЫ ТЕКСТОВОГО РЕДАКТОРА VI

В составе ОС LINUX обычно поставляются текстовые редакторы: **ed** - интерактивный строковый редактор, **vi** и **ex** - его расширенные версии. Под именем **vi** (**visual interpretator** - визуальный интерпретатор) эта программа работает как экранно-ориентированный редактор, а под именем **ex** - как строчно-ориентированный.

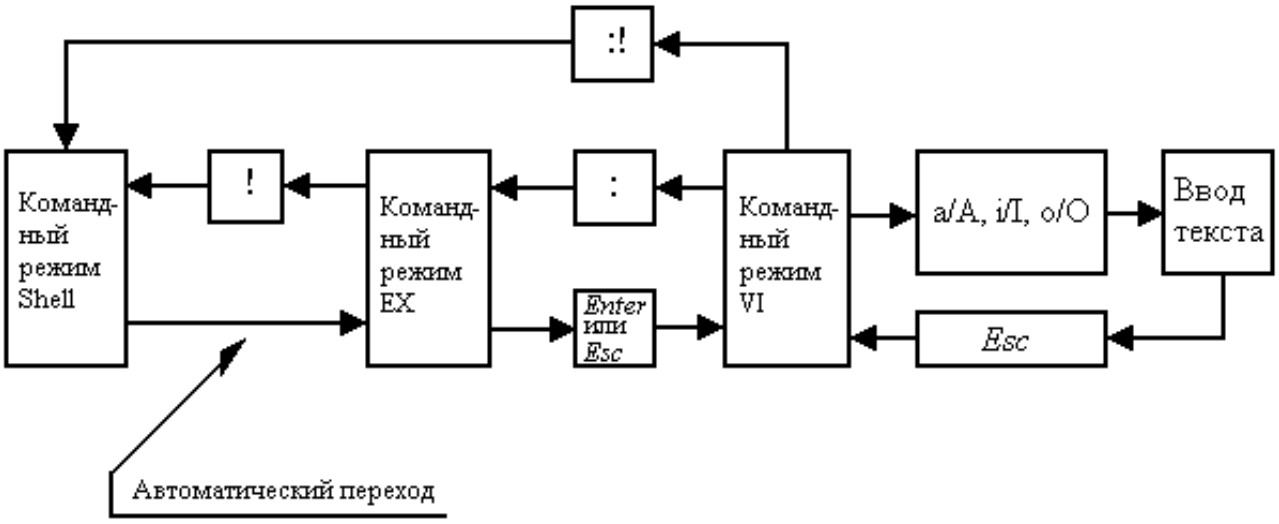
Для вызова редактора **vi** используется команда **vi:**

vi [+line] [-R] [-x] [-r] [-t] file...

где +line - номер строки, с которой Вы хотите начать редактирование; R - читать; это означает, что файл можно только просматривать, но не модифицировать; x - расшифровывающее чтение т.е. просмотр файла, зашифрованного командой **crypt**, или редактирование обычного текста с последующим шифрованием по мере записи на диск; r - восстановление файла после системного или программного крахов; t - вызов для редактирования файла, который содержит названный (в поле file команды **vi**) тег (tag). Тег - это список символов, с которых начинается раздел в текстовом файле. Теги разных файлов объединяют в один файл - файл тегов с именем tags. Опцией -t обеспечивается вызов файла tags, который содержит названный тег и имя редактируемого файла, в котором тег находится. Команду вызова редактора можно использовать в форме **vi +/word/file** - начало редактирования файла file с первой строки, которая содержит слово word, или в форме **vi +file** - начало редактирования файла с последней строки.

Структура редактора

Работая с редактором, пользователь находится или в одном из его командных режимов, или в режиме ввода текста. Ниже приведенная схема иллюстрирует взаимодействие этих режимов и способы перехода редактора между ними.



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

$ vi text

_

~

.

.

"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 *ВаскSpace*.

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;

Изучение других многочисленных команд этого командного режима целесообразно проводить, разбив их на тематические группы. Они приведены в разделе 2.2.

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, добавле-

ния 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: set sw=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 >

Составители: Лянцев Олег Дмитриевич, Казанцев Андрей Валерьевич

РАБОТА В ОПЕРАЦИОННОЙ СИСТЕМЕ LINUX

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к лабораторному практикуму по курсу

«Операционные системы» для студентов специальностей 230106 – «Применение и эксплуатация автоматизированных систем специального назначения» и 230100 «Информатика и вычислительная техника».

Подписано к печати Формат 60x84 1/16.
Бумага писчая. Печать плоская. Усл. печ. л.2,25.
Усл. кр. –отт. 2,0. Уч. –изд. л. 2,0. Тираж 100 экз.
Заказ №
Уфимский государственный авиационный технический университет
Центр оперативной полиграфии УГАТУ
450000, Уфа-центр, ул. К. Маркса, 12