

РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ

Факультет физико-математических и естественных наук

Операционные системы

Лабораторные работы

Учебное пособие

Москва

Российский университета дружбы народов

2016

УДК 004.051 (075.8)
ББК 018.2*32.973
О 60

Утверждено
РИС Учёного совета
Российского университета
дружбы народов

Рецензенты:

доцент, кандидат физико-математических наук, зав. сектором телекоммуникаций
УИТОиСТС РУДН, Ловецкий К. П.,

доцент, кандидат физико-математических наук, с.н.с. ЛИТ ОИЯИ Стрельцова О. И.

О-60 **Авторский коллектив: Д. С. Кулябов, М. Н. Геворкян,**
А. В. Королькова, А. В. Демидова.
Операционные системы: лабораторные работы : учебное пособие /
Д. С. Кулябов, М. Н. Геворкян, А. В. Королькова, А. В. Демидова. —
Москва : РУДН, 2016. — 117 с. : ил.

Данное учебное пособие рекомендуется для проведения лабораторных работ по курсу «Операционные системы» для направлений 02.03.02 «Фундаментальная информатика и информационные технологии», 02.03.01 «Математика и компьютерные науки», 38.03.05 «Бизнес-информатика», 01.03.02 «Прикладная математика и информатика», 09.03.03 «Прикладная информатика».

УДК 004.451(075.8)
ББК 018.2*32.973

ISBN 978-5-209-07626-1

© Кулябов Д. С., Геворкян М. Н.,
Королькова А. В., Демидова А. В., 2016
© Российский университет дружбы народов, 2016

Оглавление

Лабораторная работа № 1. Знакомство с операционной системой Linux .	6
1.1. Цель работы	6
1.2. Многопользовательская модель разграничения доступа операционной системы	6
1.3. Виртуальные консоли	7
1.4. Конфигурирование основных тулkitов (инструментария) операционной системы	8
1.5. Последовательность выполнения работы	10
1.6. Содержание отчёта	11
1.7. Контрольные вопросы	11
Лабораторная работа № 2. Основы интерфейса взаимодействия пользователя с системой Unix на уровне командной строки	12
2.1. Цель работы	12
2.2. Указания к работе	12
2.3. Последовательность выполнения работы	17
2.4. Содержание отчёта	17
2.5. Контрольные вопросы	18
Лабораторная работа № 3. Анализ файловой системы Linux. Команды для работы с файлами и каталогами	19
3.1. Цель работы	19
3.2. Указания к работе	19
3.3. Последовательность выполнения работы	24
3.4. Содержание отчёта	25
3.5. Контрольные вопросы	25
Лабораторная работа № 4. Поиск файлов. Перенаправление ввода-вывода. Просмотр запущенных процессов	26
4.1. Цель работы	26
4.2. Указания к работе	26
4.3. Последовательность выполнения работы	29
4.4. Содержание отчёта	29
4.5. Контрольные вопросы	29
Лабораторная работа № 5. Командная оболочка Midnight Commander .	30
5.1. Цель работы	30
5.2. Указания к работе	30
5.3. Последовательность выполнения работы	38
5.4. Содержание отчёта	39
5.5. Контрольные вопросы	39
Лабораторная работа № 6. Текстовый редактор vi	40
6.1. Цель работы	40
6.2. Указания к работе	40
6.3. Последовательность выполнения работы	43
6.4. Содержание отчёта	44
6.5. Контрольные вопросы	44

Лабораторная работа № 7. Текстовый редактор emacs	46
7.1. Цель работы	46
7.2. Указания к работе	46
7.3. Последовательность выполнения работы	50
7.4. Содержание отчёта	52
7.5. Контрольные вопросы	52
Лабораторная работа № 8. Программирование в командном процессоре	
ОС UNIX. Командные файлы	53
8.1. Цель работы	53
8.2. Указания к лабораторной работе	53
8.3. Последовательность выполнения работы	65
8.4. Содержание отчёта	66
8.5. Контрольные вопросы	66
Лабораторная работа № 9. Программирование в командном процессоре	
ОС UNIX. Ветвления и циклы	67
9.1. Цель работы	67
9.2. Последовательность выполнения работы	67
9.3. Содержание отчёта	67
9.4. Контрольные вопросы	68
Лабораторная работа № 10. Программирование в командном процес-	
соре ОС UNIX. Расширенное программирование	69
10.1. Цель работы	69
10.2. Последовательность выполнения работы	69
10.3. Содержание отчёта	69
10.4. Контрольные вопросы	69
Лабораторная работа № 11. Средства, применяемые при разработке	
программного обеспечения в ОС типа UNIX/Linux	71
11.1. Цель работы	71
11.2. Указания к лабораторной работе	71
11.3. Последовательность выполнения работы	75
11.4. Содержание отчёта	79
11.5. Контрольные вопросы	79
Лабораторная работа № 12. Управление версиями	80
12.1. Цель работы	80
12.2. Системы контроля версий. Общие понятия	80
12.3. Указания к лабораторной работе	81
12.4. Последовательность выполнения работы	84
12.5. Содержание отчёта	84
12.6. Контрольные вопросы	84
Лабораторная работа № 13. Именованные каналы	85
13.1. Цель работы	85
13.2. Указания к работе	85
13.3. Пример программы	86
13.4. Последовательность выполнения работы	89
13.5. Содержание отчёта	89
13.6. Контрольные вопросы	89

Лабораторная работа № 14. Очереди сообщений	90
14.1. Цель работы	90
14.2. Указания к работе	90
14.3. Пример программы	92
14.4. Последовательность выполнения работы	95
14.5. Содержание отчёта	95
14.6. Контрольные вопросы	96
Лабораторная работа № 15. Сокеты	97
15.1. Цель работы	97
15.2. Указания к работе	97
15.3. Пример программы	101
15.4. Последовательность выполнения работы	106
15.5. Содержание отчёта	106
15.6. Контрольные вопросы	106
Учебно-методический комплекс	107
Программа дисциплины	109
Цели и задачи дисциплины	109
Место дисциплины в структуре ООП	109
Требования к результатам освоения дисциплины	109
Объем дисциплины и виды учебной работы	110
Содержание дисциплины	110
Лабораторный практикум	112
Методические рекомендации по организации изучения дисциплины	112
Учебно-методическое и информационное обеспечение дисциплины	115
Календарный план	116
Сведения об авторах	117

Лабораторная работа № 1. Знакомство с операционной системой Linux

1.1. Цель работы

Познакомиться с операционной системой Linux, получить практические навыки работы с консолью и некоторыми графическими менеджерами рабочих столов операционной системы.

1.2. Многопользовательская модель разграничения доступа операционной системы

Linux — многопользовательская операционная система, т.е. несколько пользователей могут работать с ней одновременно с помощью терминалов.

Определение 1. *Компьютерный терминал* — устройство ввода–вывода, основные функции которого заключаются в вводе и отображении данных.

Определение 2. *Текстовый терминал (терминал, текстовая консоль)* — интерфейс компьютера для последовательной передачи данных.

Загрузка системы завершается выводом на экран приглашения пользователя к регистрации «login:». После этого система запросит *пароль (password)*, соответствующий введенному имени, выдав специальное приглашение — обычно «Password:».

```
hostname login: username
Password:
```

Процедура регистрации в системе обязательна для Linux. Каждый пользователь операционной системы имеет определенные ограничения на возможные с его стороны действия: чтение, изменение, запуск файлов, а также на ресурсы: пространство на файловой системе, процессорное время для выполнения текущих задач (процессов). При этом действия одного пользователя не влияют на работу другого. Такая модель разграничения доступа к ресурсам операционной системы получила название *многопользовательской*.

В многопользовательской модели пользователи делятся на *пользователей с обычными правами и администраторов*. Пользователь с обычными правами может производить действия с элементами операционной системы только в рамках выделенного ему пространства и ресурсов, не влияя на жизнеспособность самой операционной системы и работу других пользователей. Полномочия же пользователей с административными правами обычно не ограничены.

Для каждого пользователя организуется домашний каталог, где хранятся его данные и настройки рабочей среды. Доступ других пользователей с обычными правами к этому каталогу ограничивается.

Определение 3. *Учётная запись пользователя (user account)* — идентификатор пользователя, на основе которого ему назначаются права на действия в операционной системе.

Учётная запись пользователя содержит:

- входное имя пользователя (Login Name);
- пароль (Password);
- внутренний идентификатор пользователя (User ID);

- идентификатор группы (Group ID);
- анкетные данные пользователя (General Information);
- домашний каталог (Home Dir);
- указатель на программную оболочку (Shell).

Определение 4. *Входное имя пользователя (Login)* — название учётной записи пользователя.

Входному имени пользователя ставится в соответствие *внутренний идентификатор пользователя в системе (User ID, UID)* — положительное целое число в диапазоне от 0 до 65535, по которому в системе однозначно отслеживаются действия пользователя.

Пользователю может быть назначена определенная группа для доступа к некоторым ресурсам, разграничения прав доступа к различным файлам и директориям. Каждая группа пользователей в операционной системе имеет свой идентификатор — *Group ID (GID)*.

Анкетные данные пользователя (General Information или GECOS) являются необязательным параметром учётной записи и могут содержать реальное имя пользователя (фамилию, имя), адрес, телефон.

В домашнем каталоге пользователя хранятся данные (файлы) пользователя, настройки рабочего стола и других приложений. Содержимое домашнего каталога обычно не доступно другим пользователям с обычными правами и не влияет на работу и настройки рабочей среды других пользователей.

Учётная запись пользователя с $UID=0$ называется *root* и присутствует в любой системе типа Linux. Пользователь *root* имеет права администратора и может выполнять любые действия в системе. Работать под учётной записью *root* следует только тогда, когда это действительно необходимо: при настройке и обновлении системы, восстановлении после сбоев.

Учётные записи пользователей хранятся в файле `/etc/passwd`, который имеет следующую структуру:

```
login:password:UID:GID:GECOS:home:shell
```

Например, учётные записи пользователей *root* и *ivan* в файле `/etc/passwd` могут быть записаны следующим образом:

```
root:x:0:0:root:/root:/bin/bash
ivan:x:1000:100::/home/ivan:/bin/bash
```

Замечание 1. Изначально поле пароля содержало хеш пароля и использовалось для аутентификации. Однако из соображений безопасности все пароли были перенесены в специальный файл `/etc/shadow`, недоступный для чтения обычным пользователям. Поэтому в файле `/etc/passwd` поле `password` имеет значение `x`.

Замечание 2. Символ `*` в поле `password` некоторой учётной записи в файле `/etc/passwd` означает, что пользователь не сможет войти в систему.

1.3. Виртуальные консоли

Определение 5. *Виртуальные консоли* — реализация концепции многотерминальной работы в рамках одного устройства.

В операционных системах типа Linux доступно обычно 6 виртуальных консолей, работающих в текстовом режиме. Переключение между консолями осуществляется при помощи сочетания клавиши `[Alt]` с одной из функциональных клавиш

(F1–F6)). Виртуальные консоли при обращении к ним из командной строки обозначаются `ttyN`, где N — номер виртуальной консоли.

Для перехода из текстового режима в графический необходимо нажать комбинацию клавиш **Ctrl** + **Alt** + **F7**. Для переключения из графического режима в одну из текстовых виртуальных консолей достаточно нажать комбинацию клавиш **Ctrl** + **Alt** + **F_n**, где *n* — номер необходимой виртуальной консоли. Процедура регистрации в графическом режиме аналогична регистрации в текстовом режиме.

Если пользователь входит в систему несколько раз под одним и тем же именем (на разных виртуальных консолях), то ему будут доступны несколько разных сеансов работы, не связанных между собой.

Для корректного завершения своей работы в системе пользователь должен выйти из системы. Чтобы завершить работу в виртуальной консоли, пользователю необходимо в соответствующей командной строке набрать команду `logout` или воспользоваться комбинацией клавиш **Ctrl** + **D**. При этом работа самой операционной системы не прерывается.

Определение 6. Весь процесс взаимодействия пользователя с системой с момента регистрации до выхода называется *сеансом работы*.

1.4. Конфигурирование основных тулкитов (инструментария) операционной системы

Определение 7. *Toolkit* (*Тк*, «набор инструментов», «инструментарий») — кроссплатформенная библиотека базовых элементов графического интерфейса, распространяемая с открытыми исходными текстами.

Используются следующие основные тулкиты:

- GTK+ (сокращение от GIMP Toolkit) — кроссплатформенная библиотека элементов интерфейса;
- Qt — кросс-платформенный инструментарий разработки программного обеспечения на языке программирования C++.

GTK+ состоит из двух компонентов:

- GTK — содержит набор элементов пользовательского интерфейса (таких, как кнопка, список, поле для ввода текста и т. п.) для различных задач;
- GDK — отвечает за вывод информации на экран, может использовать для этого X Window System, Linux Framebuffer, WinAPI.

На основе GTK+ построены рабочие окружения GNOME, LXDE и Xfce.

Естественно, эти тулкиты могут использоваться и за пределами «родных» десктопных окружений.

Qt используется в среде KDE (Kool Desktop Environment).

1.4.1. Выбор графической среды при логине

На компьютерах с операционной системой типа Linux может быть установлено несколько графических сред. После загрузки компьютера появится менеджер дисплея (рис. 1.1).

Кратко опишем некоторые из доступных графических сред.

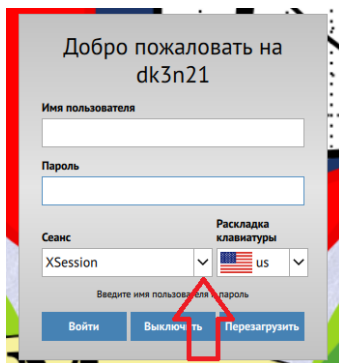


Рис. 1.1. Менеджер дисплеев. Красной стрелкой обозначен выпадающий список, позволяющий выбрать графическую среду для загрузки

1.4.2. Среда Xfce

Рабочая среда Xfce легковесна, построена по модульному принципу и позволяет гибко настраивать рабочее пространство пользователя. Xfce основана на GTK+ версии 2 и использует менеджер окон Xfwm. Начиная с версии 4.6, настройки рабочей среды хранятся в реестре `xfconf`.

Элементы Xfce:

- файловый менеджер Thunar;
- менеджер окон Xfwm;
- панель задач `xfce4-panel`;
- менеджер рабочего стола `xfdesktop`;
- менеджер сеансов `xfce4-session`;
- диспетчер настроек `xfce4-settings`;
- система хранения настроек `xfconf`;
- поиск приложений `xfce4-appfinder`;
- эмулятор терминала `xfce4-terminal`;
- менеджер питания `xfce4-power-manager`.

1.4.3. Среда GNOME

Для хранения системных настроек среды GNOME, начиная с версии 3.0, используется фреймворк GSettings, основанный на формате файлов `dconf`. Для настройки рабочего пространства GNOME используется утилита `gnome-control-center`. Это своеобразный центр управления, позволяющий установить параметры среды, оформление, произвести настройки разных программ.

Следует заметить, что установки оформления будут действовать только в рамках самой среды GNOME, и если вы желаете использовать GTK-приложения в рамках других сред, следует провести конфигурирование GTK отдельно.

Некоторые элементы GNOME:

- файловый менеджер Nautilus;
- эмулятор терминала GNOME Terminal;
- текстовый редактор gedit;

- приложение для просмотра документации Yelp;
- стандартный веб-браузер Web (ранее — Epiphany);
- приложение для управления электронной почтой Evolution;
- комплект графических средств для администрирования GNOME System Tools.

1.4.4. Среда KDE

Для конфигурирования данной среды следует использовать утилиту:

```
systemsettings5
```

Некоторые элементы KDE:

- базовые библиотеки KDELibs;
- компонент для просмотра HTML документов KHTML;
- компонент, обеспечивающий доступ к файлам KIO;
- оконный менеджер KWin;
- рабочий стол и основные приложения kdesktop;
- инструменты графического администрирования kadmin;
- утилиты kdeutils.

1.5. Последовательность выполнения работы

Замечание. Скриншоты можно сделать с помощью команды `import screen.png`. После ввода этой команды курсор мыши превратится в перекрестие, которым можно выделить желаемую область экрана для сохранения в файл `screen.png`. Не забывайте каждый раз изменять название файла при создании нескольких снимков экрана.

1. Ознакомиться с теоретическим материалом.
2. Загрузить компьютер.
3. Перейти на текстовую консоль. Сколько текстовых консолей доступно на вашем компьютере?
4. Перемещаться между текстовыми консолями. Какие комбинации клавиш необходимо при этом нажимать?
5. Зарегистрироваться в *текстовой* консоли операционной системы. Какой логин вы при этом использовали? Какие символы отображаются при вводе пароля?
6. Завершить консольный сеанс. Какую команду или комбинацию клавиш необходимо для этого использовать?
7. Переключиться на графический интерфейс. Какую комбинацию клавиш для этого необходимо нажать?
8. Ознакомиться с менеджером рабочих столов. Как называется менеджер, запускаемый по умолчанию?
9. Поочередно зарегистрироваться в разных графических менеджерах рабочих столов (GNOME, KDE, XFCE) и оконных менеджерах (Openbox). Продемонстрировать разницу между ними, сделав снимки экрана (скриншоты). Какие графические менеджеры установлены на вашем компьютере?
10. Изучить список установленных программ. Обратить внимание на предпочтительные программы для разных применений. Запустите поочередно браузер, текстовый редактор, текстовый процессор, эмулятор консоли. Укажите названия программ.

1.6. Содержание отчёта

1. Титульный лист с указанием номера лабораторной работы и ФИО студента.
2. Формулировка цели работы.
3. Описание результатов выполнения задания:
 - скриншоты (снимки экрана), фиксирующие выполнение лабораторной работы;
 - листинги (исходный код) программ (если они есть);
 - результаты выполнения программ (текст или снимок экрана в зависимости от задания).
4. Выводы, согласованные с целью работы.
5. Ответы на контрольные вопросы.

1.7. Контрольные вопросы

1. Что такое компьютерный терминал? Есть ли, по вашему мнению, у него преимущества перед графическим интерфейсом?
2. Что такое входное имя пользователя?
3. В каком файле хранятся пароли пользователей? В каком виде они хранятся?
4. Где хранятся настройки пользовательских программ?
5. Какое входное имя у администратора ОС Unix?
6. Имеет ли администратор доступ к настройкам пользователей?
7. Каковы основные характеристики многопользовательской модели разграничения доступа?
8. Какую информацию кроме пароля и логина содержит учётная запись пользователя?
9. Что такое UID и GID? Расшифруйте эти аббревиатуры.
10. Что такое GECOS?
11. Что такое домашний каталог? Какие файлы хранятся в нем?
12. Как называется ваш домашний каталог?
13. Имеет ли администратор возможность изменить содержимое домашнего каталога пользователя?
14. Что хранится в файле `/etc/passwd`?
15. Как, просмотрев содержимое файла `/etc/passwd`, узнать, какие пользователи не смогут войти в систему?
16. Что такое виртуальные консоли? Как вы думаете, что означает слово «виртуальный» в данном контексте?
17. Зачем нужна программа `getty`?
18. Что такое сеанс работы?
19. Что такое тулkit?
20. Какие основные тулкиты существуют в системе Unix?

Лабораторная работа № 2. Основы интерфейса взаимодействия пользователя с системой Unix на уровне командной строки

2.1. Цель работы

Приобретение практических навыков взаимодействия пользователя с системой посредством командной строки.

2.2. Указания к работе

В операционной системе типа Linux взаимодействие пользователя с системой обычно осуществляется с помощью командной строки посредством построчного ввода команд. При этом обычно используются командные интерпретаторы языка shell: /bin/sh; /bin/csh; /bin/ksh.

Формат команды. Командой в операционной системе называется записанный по специальным правилам текст (возможно с аргументами), представляющий собой указание на выполнение какой-либо функций (или действий) в операционной системе. Обычно первым словом идёт имя команды, остальной текст — аргументы или опции, конкретизирующие действие.

Общий формат команд можно представить следующим образом:

<имя_команды><разделитель><аргументы>

Команда man. Команда man используется для просмотра (оперативная помощь) в диалоговом режиме руководства (manual) по основным командам операционной системы типа Linux.

Формат команды:

man <команда>

Пример (вывод информации о команде man):

man man

Для управления просмотром результата выполнения команды man можно использовать следующие клавиши:

- **Space** — перемещение по документу на одну страницу вперёд;
- **Enter** — перемещение по документу на одну строку вперёд;
- **q** — выход из режима просмотра описания.

Команда cd. Команда cd используется для перемещения по файловой системе операционной системы типа Linux.

Замечание 3. Файловая система ОС типа Linux — иерархическая система каталогов, подкаталогов и файлов, которые обычно организованы и сгруппированы по функциональному признаку. Самый верхний каталог в иерархии называется корневым и обозначается символом /. Корневой каталог содержит системные файлы и другие каталоги.

Формат команды:

cd [путь_к_каталогу]

Для перехода в домашний каталог пользователя следует использовать команду `cd` без параметров или `cd ~`.

Например, команда

```
cd /afs/dk.sci.pfu.edu.ru/home
```

позволяет перейти в каталог `/afs/dk.sci.pfu.edu.ru/home` (если такой существует), а для того, чтобы подняться выше на одну директорию, следует использовать:

```
cd ..
```

Подробнее об опциях команды `cd` смотри в справке с помощью команды `man`:

```
man cd
```

Команда `pwd`. Для определения абсолютного пути к текущему каталогу используется команда `pwd` (`print working directory`).

Пример (абсолютное имя текущего каталога пользователя `dharm`):

```
pwd
```

результат:

```
/afs/dk.sci.pfu.edu.ru/home/d/h/dharma
```

Сокращения имён файлов. В работе с командами, в качестве аргументов которых выступает путь к какому-либо каталогу или файлу, можно использовать сокращённую запись пути. Символы сокращения приведены в табл. 2.1.

Таблица 2.1

Символы сокращения имён файлов

Символ	Значение
~	Домашний каталог
.	Текущий каталог
..	Родительский каталог

Например, в команде `cd` для перемещения по файловой системе сокращённую запись пути можно использовать следующим образом (команды чередуются с выводом результата выполнения команды `pwd`):

```
pwd
```

```
/afs/dk.sci.pfu.edu.ru/home/d/h/dharma
```

```
cd ..
```

```
pwd
```

```
/afs/dk.sci.pfu.edu.ru/home/d/h
```

```
cd ../..
```

```
pwd
```

```
/afs/dk.sci.pfu.edu.ru/home
```

```
cd ~/work
```

```
pwd
```

```
/afs/dk.sci.pfu.edu.ru/home/d/h/dharma/work
```

Команда ls. Команда `ls` используется для просмотра содержимого каталога.

Формат команды:

```
ls [-опции] [путь]
```

Пример:

```
cd
```

```
cd ..
```

```
pwd
```

```
/afs/dk.sci.pfu.edu.ru/home/d/h
```

```
ls
```

```
dharma
```

Некоторые файлы в операционной системе скрыты от просмотра и обычно используются для настройки рабочей среды. Имена таких файлов начинаются с точки. Для того, чтобы отобразить имена скрытых файлов, необходимо использовать команду `ls` с опцией `a`:

```
ls -a
```

Можно также получить информацию о типах файлов (каталог, исполняемый файл, ссылка), для чего используется опция `F`. При использовании этой опции в поле имени выводится символ, который определяет тип файла (см. табл. 2.2)

Таблица 2.2

Символ, который определяет тип файла

Тип файла	Символ
Каталог	/
Исполняемый файл	*
Ссылка	@

Чтобы вывести на экран подробную информацию о файлах и каталогах, необходимо использовать опцию `l`. При этом о каждом файле и каталоге будет выведена следующая информация:

- тип файла,
- право доступа,
- число ссылок,
- владелец,
- размер,
- дата последней ревизии,
- имя файла или каталога.

Пример:

```
cd /
```

```
ls
```

Результат:

```
bin boot dev etc home lib media mnt
opt proc root sbin sys tmp usr var
```

В этом же каталоге команда

```
ls -alF
```

даст примерно следующий результат:

```
drwxr-xr-x 21 root root 4096 Jan. 17 09:00 ./
drwxr-xr-x 21 root root 4096 Jan. 17 09:00 ../
drwxr-xr-x 2 root root 4096 Jan. 18 15:57 bin/
drwxr-xr-x 2 root root 4096 Apr. 14 2008 boot/
drwxr-xr-x 20 root root 14120 Feb. 17 10:48 dev/
drwxr-xr-x 170 root root 12288 Feb. 17 09:19 etc/
drwxr-xr-x 6 root root 4096 Aug. 5 2009 home/
lrwxrwxrwx 1 root root 5 Jan. 12 22:01 lib -> lib64/
drwxr-xr-x 8 root root 4096 Jan. 30 21:41 media/
drwxr-xr-x 5 root root 4096 Jan. 17 2010 mnt/
drwxr-xr-x 25 root root 4096 Jan. 16 09:55 opt/
dr-xr-xr-x 163 root root 0 Feb. 17 13:17 proc/
drwxr-xr-x 31 root root 4096 Feb. 15 23:57 root/
drwxr-xr-x 2 root root 12288 Jan. 18 15:57/sbin/
drwxr-xr-x 12 root root 0 Feb. 17 13:17 sys/
drwxrwxrwt 12 root root 500 Feb. 17 16:35 tmp/
drwxr-xr-x 22 root root 4096 Jan. 18 09:26 usr/
drwxr-xr-x 17 root root 4096 Jan. 14 17:38 var/
```

Команда mkdir. Команда `mkdir` используется для создания каталогов.

Формат команды:

```
mkdir имя_каталога1 [имя_каталога2...]
```

Пример создания каталога в текущем каталоге:

```
cd
pwd
```

```
/afs/dk.sci.pfu.edu.ru/home/d/h/dharma
```

```
ls
```

```
Desktop public tmp
GNUstep public_html work
```

```
mkdir abc
ls
```

```
abc GNUstep public_html work
Desktop public tmp
```

Замечание 4. Для того чтобы создать каталог в определённом месте файловой системы, должны быть правильно установлены права доступа.

Можно создать также подкаталог в существующем подкаталоге:

```
mkdir parentdir
mkdir parentdir/dir
```

При задании нескольких аргументов создаётся несколько каталогов:

```
cd parentdir
mkdir dir1 dir2 dir3
```

Можно использовать группировку:

```
mkdir parentdir/{dir1,dir2,dir3}
```

Если же требуется создать подкаталог в каталоге, отличном от текущего, то путь к нему требуется указать в явном виде:

```
mkdir ../dir1/dir2
```

или

```
mkdir ~/dir1/dir2
```

Интересны следующие опции:

--mode (или -m) — установка атрибутов доступа;

--parents (или -p) — создание каталога вместе с родительскими по отношению к нему каталогами.

Атрибуты задаются в численной или символьной нотации:

```
mkdir --mode=777 dir
```

или

```
mkdir -m a+rwX dir
```

Опция --parents (краткая форма -p) позволяет создавать иерархическую цепочку подкаталогов, создавая все промежуточные каталоги:

```
mkdir -p ~/dir1/dir2/dir3
```

Команда rm. Команда rm используется для удаления файлов и/или каталогов.

Формат команды:

```
rm [-опции] [файл]
```

Если требуется, чтобы выдавался запрос подтверждения на удаление файла, то необходимо использовать опцию i.

Чтобы удалить каталог, содержащий файлы, нужно использовать опцию r. Без указания этой опции команда не будет выполняться.

Пример:

```
cd
```

```
mkdir abs
```

```
rm abc
```

```
rm: abc is a directory
```

```
rm -r abc
```

Если каталог пуст, то можно воспользоваться командой rmdir. Если удаляемый каталог содержит файлы, то команда не будет выполнена — нужно использовать rm -r имя_каталога.

Команда history. Для вывода на экран списка ранее выполненных команд используется команда history. Выводимые на экран команды в списке нумеруются. К любой команде из выведенного на экран списка можно обратиться по её номеру в списке, воспользовавшись конструкцией !<номер_команды>.

Пример:

```
history
```

```
1 pwd
```

```
2 ls
```

```
3 ls -a
```

```
4 ls -l
```

```
5 cd /
```

```
6 history
```

```
!5
```

```
cd /
```

Можно модифицировать команду из выведенного на экран списка при помощи следующей конструкции:

```
!<номер_команды>:s/<что_меняем>/<на_что_меняем>
```

Пример:


```
!3:s/a/F
ls -F
```

Замечание 5. Если в заданном контексте встречаются специальные символы (типа «.», «/», «*» и т.д.), надо перед ними поставить символ экранирования \ (обратный слэш).

Использование символа «;». Если требуется выполнить последовательно несколько команд, записанный в одной строке, то для этого используется символ точка с запятой

Пример:

```
cd; ls
```

2.3. Последовательность выполнения работы

1. Определите полное имя вашего домашнего каталога. Далее относительно этого каталога будут выполняться последующие упражнения.
2. Выполните следующие действия:
 - 2.1. Перейдите в каталог /tmp.
 - 2.2. Выведите на экран содержимое каталога /tmp. Для этого используйте команду `ls` с различными опциями. Поясните разницу в выводимой на экран информации.
 - 2.3. Определите, есть ли в каталоге /var/spool подкаталог с именем `cron`?
 - 2.4. Перейдите в Ваш домашний каталог и выведите на экран его содержимое. Определите, кто является владельцем файлов и подкаталогов?
3. Выполните следующие действия:
 - 3.1. В домашнем каталоге создайте новый каталог с именем `newdir`.
 - 3.2. В каталоге `~/newdir` создайте новый каталог с именем `morefun`.
 - 3.3. В домашнем каталоге создайте одной командой три новых каталога с именами `letters`, `memos`, `misk`. Затем удалите эти каталоги одной командой.
 - 3.4. Попробуйте удалить ранее созданный каталог `~/newdir` командой `rm`. Проверьте, был ли каталог удалён.
 - 3.5. Удалите каталог `~/newdir/morefun` из домашнего каталога. Проверьте, был ли каталог удалён.
4. С помощью команды `man` определите, какую опцию команды `ls` нужно использовать для просмотра содержимое не только указанного каталога, но и подкаталогов, входящих в него.
5. С помощью команды `man` определите набор опций команды `ls`, позволяющий отсортировать по времени последнего изменения выводимый список содержимого каталога с развёрнутым описанием файлов.
6. Используйте команду `man` для просмотра описания следующих команд: `cd`, `pwd`, `mkdir`, `rmdir`, `rm`. Поясните основные опции этих команд.
7. Используя информацию, полученную при помощи команды `history`, выполните модификацию и исполнение нескольких команд из буфера команд.

2.4. Содержание отчёта

1. Титульный лист с указанием номера лабораторной работы и ФИО студента.
2. Формулировка цели работы.
3. Описание результатов выполнения задания:

- скриншоты (снимки экрана), фиксирующие выполнение лабораторной работы;
 - листинги (исходный код) программ (если они есть);
 - результаты выполнения программ (текст или снимок экрана в зависимости от задания).
4. Выводы, согласованные с целью работы.
 5. Ответы на контрольные вопросы.

2.5. Контрольные вопросы

1. Что такое командная строка?
2. При помощи какой команды можно определить абсолютный путь текущего каталога? Приведите пример.
3. При помощи какой команды и каких опций можно определить только тип файлов и их имена в текущем каталоге? Приведите примеры.
4. Каким образом отобразить информацию о скрытых файлах? Приведите примеры.
5. При помощи каких команд можно удалить файл и каталог? Можно ли это сделать одной и той же командой? Приведите примеры.
6. Каким образом можно вывести информацию о последних выполненных пользователем командах? работы?
7. Как воспользоваться историей команд для их модифицированного выполнения? Приведите примеры.
8. Приведите примеры запуска нескольких команд в одной строке.
9. Дайте определение и приведите примера символов экранирования.
10. Охарактеризуйте вывод информации на экран после выполнения команды `ls -l` с опцией `-l`.
11. Что такое относительный путь к файлу? Приведите примеры использования относительного и абсолютного пути при выполнении какой-либо команды.
12. Как получить информацию об интересующей вас команде?
13. Какая клавиша или комбинация клавиш служит для автоматического дополнения вводимых команд?

Лабораторная работа № 3. Анализ файловой системы Linux. Команды для работы с файлами и каталогами

3.1. Цель работы

Ознакомление с файловой системой Linux, её структурой, именами и содержанием каталогов. Приобретение практических навыков по применению команд для работы с файлами и каталогами, по управлению процессами (и работами), по проверке использования диска и обслуживанию файловой системы.

3.2. Указания к работе

3.2.1. Команды для работы с файлами и каталогами

Для создания текстового файла можно использовать команду `touch`.

Формат команды:

`touch имя-файла`

Для просмотра файлов небольшого размера можно использовать команду `cat`.

Формат команды:

`cat имя-файла`

Для просмотра файлов постранично удобнее использовать команду `less`.

Формат команды:

`less имя-файла`

Следующие клавиши используются для управления процессом просмотра:

- `Space` — переход к следующей странице,
- `ENTER` — сдвиг вперёд на одну строку,
- `b` — возврат на предыдущую страницу,
- `h` — обращение за подсказкой,
- `q` — выход из режима просмотра файла.

Команда `head` выводит по умолчанию первые 10 строк файла.

Формат команды:

`head [-n] имя-файла,`

где `n` — количество выводимых строк.

Команда `tail` выводит по умолчанию 10 последних строк файла.

Формат команды:

`tail [-n] имя-файла,`

где `n` — количество выводимых строк.

3.2.2. Копирование файлов и каталогов

Команда `cp` используется для копирования файлов и каталогов.

Формат команды:

`cp [-опции] исходный_файл целевой_файл`

Примеры:

1. Копирование файла в текущем каталоге. Скопировать файл `~/abc1` в файл `april` и в файл `may`:

```
cd
touch abc1
cp abc1 april
cp abc1 may
```

2. Копирование нескольких файлов в каталог. Скопировать файлы `april` и `may` в каталог `monthly`:

```
mkdir monthly
cp april may monthly
```
3. Копирование файлов в произвольном каталоге. Скопировать файл `monthly/may` в файл с именем `june`:

```
cp monthly/may monthly/june
ls monthly
```

Опция `i` в команде `cp` выведет на экран запрос подтверждения о перезаписи файла.

Для рекурсивного копирования каталогов, содержащих файлы, используется команда `cp` с опцией `r`.

Примеры:

1. Копирование каталогов в текущем каталоге. Скопировать каталог `monthly` в каталог `monthly.00`:

```
mkdir monthly.00
cp -r monthly monthly.00
```
2. Копирование каталогов в произвольном каталоге. Скопировать каталог `monthly.00` в каталог `/tmp`

```
cp -r monthly.00 /tmp
```

3.2.3. Перемещение и переименование файлов и каталогов

Команды `mv` и `mkdir` предназначены для перемещения и переименования файлов и каталогов.

Формат команды `mv`:

```
mv [-опции] старый_файл новый_файл
```

Примеры:

1. Переименование файлов в текущем каталоге. Изменить название файла `april` на `july` в домашнем каталоге:

```
cd
mv april july
```
2. Перемещение файлов в другой каталог. Переместить файл `july` в каталог `monthly.00`:

```
mv july monthly.00
ls monthly.00
```

Результат:

```
april july june may
```

Если необходим запрос подтверждения о перезаписи файла, то нужно использовать опцию `i`.

3. Переименование каталогов в текущем каталоге. Переименовать каталог `monthly.00` в `monthly.01`

```
mv monthly.00 monthly.01
```
4. Перемещение каталога в другой каталог. Переместить каталог `monthly.01` в каталог `reports`:

```
mkdir reports
mv monthly.01 reports
```

5. Переименование каталога, не являющегося текущим. Переименовать каталог `reports/monthly.01` в `reports/monthly`:
- ```
mv reports/monthly.01 reports/monthly
```

### 3.2.4. Права доступа

Каждый файл или каталог имеет права доступа (табл. 3.1).

Таблица 3.1

| Права доступа |             |                                                      |                                                                  |
|---------------|-------------|------------------------------------------------------|------------------------------------------------------------------|
| Право         | Обозначение | Файл                                                 | Каталог                                                          |
| Чтение        | r           | Разрешены просмотр и копирование                     | Разрешён просмотр списка входящих файлов                         |
| Запись        | w           | Разрешены изменение и переименование                 | Разрешены создание и удаление файлов                             |
| Выполнение    | x           | Разрешено выполнение файла (скриптов и/или программ) | Разрешён доступ в каталог и есть возможность сделать его текущим |

В сведениях о файле или каталоге указываются:

- тип файла (символ (–) обозначает файл, а символ (d) — каталог);
- права для владельца файла (r — разрешено чтение, w — разрешена запись, x — разрешено выполнение, – — право доступа отсутствует);
- права для членов группы (r — разрешено чтение, w — разрешена запись, x — разрешено выполнение, – — право доступа отсутствует);
- права для всех остальных (r — разрешено чтение, w — разрешена запись, x — разрешено выполнение, – — право доступа отсутствует).

**Примеры:**

1. Для файла (крайнее левое поле имеет значение –) владелец файла имеет право на чтение и запись (rw–), группа, в которую входит владелец файла, может читать файл (r--), все остальные могут читать файл (r--):  
–rw-r--r--
2. Только владелец файла имеет право на чтение, изменение и выполнение файла:  
–rwx-----
3. Владелец каталога (крайнее левое поле имеет значение d) имеет право на просмотр, изменение и доступа в каталог, члены группы могут входить и просматривать его, все остальные — только входить в каталог:  
drwxr-x--x

### 3.2.5. Изменение прав доступа

Права доступа к файлу или каталогу можно изменить, воспользовавшись командой `chmod`. Сделать это может владелец файла (или каталога) или пользователь с правами администратора.

Формат команды:

chmod режим имя\_файла

Режим (в формате команды) имеет следующие компоненты структуры и способ записи:

= установить право

- лишить права

+ дать право

**r** чтение

**w** запись

**x** выполнение

**u (user)** владелец файла

**g (group)** группа, к которой принадлежит владелец файла

**o (others)** все остальные

В работе с правами доступа можно использовать их цифровую запись (восьмеричное значение) вместо символьной (табл. 3.2).

Таблица 3.2

Формы записи прав доступа

| Двоичная | Восьмеричная | Символьная |
|----------|--------------|------------|
| 111      | 7            | rwx        |
| 110      | 6            | rw-        |
| 101      | 5            | r-x        |
| 100      | 4            | r--        |
| 011      | 3            | -wx        |
| 010      | 2            | -w-        |
| 001      | 1            | --x        |
| 000      | 0            | ---        |

### Примеры:

1. Требуется создать файл ~/may с правом выполнения для владельца:

```
cd
touch may
ls -l may
chmod u+x may
ls -l may
```

2. Требуется лишить владельца файла ~/may права на выполнение:

```
chmod u-x may
ls -l may
```

3. Требуется создать каталог monthly с запретом на чтение для членов группы и всех остальных пользователей:

```
cd
mkdir monthly
chmod g-r, o-r monthly
```

4. Требуется создать файл ~/abc1 с правом записи для членов группы:

```
cd
touch abc1
chmod g+w abc1
```

### 3.2.6. Анализ файловой системы

Файловая система в Linux состоит из фалов и каталогов. Каждому физическому носителю соответствует своя файловая система.

Существует несколько типов файловых систем. Перечислим наиболее часто встречающиеся типы:

- ext2fs (second extended filesystem);
- ext2fs (third extended file system);
- ext4 (fourth extended file system);
- ReiserFS;
- xfs;
- fat (file allocation table);
- ntfs (new technology file system).

Для просмотра используемых в операционной системе файловых систем можно воспользоваться командой `mount` без параметров. В результате её применения можно получить примерно следующее:

`mount`

```
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec)
udev on /dev type tmpfs (rw,nosuid)
devpts on /dev/pts type devpts (rw,nosuid,noexec)
/dev/sda1 on /mnt/a type ext3 (rw,noatime)
/dev/sdb2 on /mnt/docs type reiserfs (rw,noatime)
shm on /dev/shm type tmpfs (rw,noexec,nosuid,nodev)
usbfs on /proc/bus/usb type usbfs
 (rw,noexec,nosuid,devmode=0664,devgid=85)
binfmt_misc on /proc/sys/fs/binfmt_misc type binfmt_misc
 (rw,noexec,nosuid,nodev)
nfsd on /proc/fs/nfs type nfsd (rw,noexec,nosuid,nodev)
```

В данном случае указаны имена устройств, названия соответствующих им точек монтирования (путь), тип файловой системы и параметрами монтирования.

В контексте команды `mount` *устройство* — специальный файл устройства, с помощью которого операционная система получает доступ к аппаратному устройству. Файлы устройств обычно располагаются в каталоге `/dev`, имеют сокращённые имена (например, `sdaN`, `sdbN` или `hdaN`, `hdbN`, где `N` — порядковый номер устройства, `sd` — устройства SCSI, `hd` — устройства MFM/IDE).

*Точка монтирования* — каталог (путь к каталогу), к которому присоединяются файлы устройств.

Другой способ определения смонтированных в операционной системе файловых систем — просмотр файла `/etc/fstab`. Сделать это можно например с помощью команды `cat`:

`cat /etc/fstab`

```
/dev/hda1 / ext2 defaults 1 1
/dev/hda5 /home ext2 defaults 1 2
/dev/hda6 swap swap defaults 0 0
/dev/hdc /mnt/cdrom auto umask=0,user,noauto,ro,exec,users 0 0
none /mnt/floppy supermount dev=/dev/fd0,fs=ext2:vfat,--,
 sync,umask=0 0 0
none /proc proc defaults 0 0
none /dev/pts devpts mode=0622 0 0
```

В каждой строке этого файла указано:

- имя устройство;
  - точка монтирования;
  - тип файловой системы;
  - опции монтирования;
  - специальные флаги для утилиты `dump`;
  - порядок проверки целостности файловой системы с помощью утилиты `fsck`.
- Для определения объёма свободного пространства на файловой системе можно воспользоваться командой `df`, которая выведет на экран список всех файловых систем в соответствии с именами устройств, с указанием размера и точки монтирования. Например:

```
df
```

```
Filesystem 1024-blocks Used Available Capacity Mounted on
/dev/hda3 297635 169499 112764 60% /
```

С помощью команды `fsck` можно проверить (а в ряде случаев восстановить) целостность файловой системы:

Формат команды:

```
fsck имя_устройства
```

Пример:

```
fsck /dev/sda1
```

### 3.3. Последовательность выполнения работы

1. Выполните все примеры, приведённые в первой части описания лабораторной работы.
2. Выполните следующие действия, зафиксировав в отчёте по лабораторной работе используемые при этом команды и результаты их выполнения:
  - 2.1. Скопируйте файл `/usr/include/sys/io.h` в домашний каталог и назовите его `equipment`. Если файла `io.h` нет, то используйте любой другой файл в каталоге `/usr/include/sys/` вместо него.
  - 2.2. В домашнем каталоге создайте директорию `~/ski.places`.
  - 2.3. Переместите файл `equipment` в каталог `~/ski.places`.
  - 2.4. Переименуйте файл `~/ski.places/equipment` в `~/ski.places/equiplist`.
  - 2.5. Создайте в домашнем каталоге файл `abc1` и скопируйте его в каталог `~/ski.places`, назовите его `equiplist2`.
  - 2.6. Создайте каталог с именем `equipment` в каталоге `~/ski.places`.
  - 2.7. Переместите файлы `~/ski.places/equiplist` и `equiplist2` в каталог `~/ski.places/equipment`.
  - 2.8. Создайте и переместите каталог `~/newdir` в каталог `~/ski.places` и назовите его `plans`.
3. Определите опции команды `chmod`, необходимые для того, чтобы присвоить перечисленным ниже файлам выделенные права доступа, считая, что в начале таких прав нет:
  - 3.1. `drwxr--r--` ... `australia`
  - 3.2. `drwx--x--x` ... `play`
  - 3.3. `-r-xr--r--` ... `my_os`
  - 3.4. `-rw-rw-r--` ... `feathers`

При необходимости создайте нужные файлы.
4. Прodelайте приведённые ниже упражнения, записывая в отчёт по лабораторной работе используемые при этом команды:



- 4.1. Просмотрите содержимое файла `/etc/password`.
- 4.2. Скопируйте файл `~/feathers` в файл `~/file.old`.
- 4.3. Переместите файл `~/file.old` в каталог `~/play`.
- 4.4. Скопируйте каталог `~/play` в каталог `~/fun`.
- 4.5. Переместите каталог `~/fun` в каталог `~/play` и назовите его `games`.
- 4.6. Лишите владельца файла `~/feathers` права на чтение.
- 4.7. Что произойдёт, если вы попытаетесь просмотреть файл `~/feathers` командой `cat`?
- 4.8. Что произойдёт, если вы попытаетесь скопировать файл `~/feathers`?
- 4.9. Дайте владельцу файла `~/feathers` право на чтение.
- 4.10. Лишите владельца каталога `~/play` права на выполнение.
- 4.11. Перейдите в каталог `~/play`. Что произошло?
- 4.12. Дайте владельцу каталога `~/play` право на выполнение.
5. Прочитайте `man` по командам `mount`, `fsck`, `mkfs`, `kill` и кратко их охарактеризуйте, приведя примеры.

### 3.4. Содержание отчёта

1. Титульный лист с указанием номера лабораторной работы и ФИО студента.
2. Формулировка цели работы.
3. Описание результатов выполнения задания:
  - скриншоты (снимки экрана), фиксирующие выполнение лабораторной работы;
  - листинги (исходный код) программ (если они есть);
  - результаты выполнения программ (текст или снимок экрана в зависимости от задания).
4. Выводы, согласованные с целью работы.
5. Ответы на контрольные вопросы.

### 3.5. Контрольные вопросы

1. Дайте характеристику каждой файловой системе, существующей на жёстком диске компьютера, на котором вы выполняли лабораторную работу.
2. Приведите общую структуру файловой системы и дайте характеристику каждой директории первого уровня этой структуры.
3. Какая операция должна быть выполнена, чтобы содержимое некоторой файловой системы было доступно операционной системе?
4. Назовите основные причины нарушения целостности файловой системы. Как устранить повреждения файловой системы?
5. Как создаётся файловая система?
6. Дайте характеристику командам для просмотра текстовых файлов.
7. Приведите основные возможности команды `cp` в Linux.
8. Приведите основные возможности команды `mv` в Linux.
9. Что такое права доступа? Как они могут быть изменены?

При ответах на вопросы используйте дополнительные источники информации по теме.

## Лабораторная работа № 4. Поиск файлов. Перенаправление ввода-вывода. Просмотр запущенных процессов

### 4.1. Цель работы

Ознакомление с инструментами поиска файлов и фильтрации текстовых данных. Приобретение практических навыков: по управлению процессами (и заданиями), по проверке использования диска и обслуживанию файловых систем.

### 4.2. Указания к работе

#### 4.2.1. Перенаправление ввода-вывода

В системе по умолчанию открыто три специальных потока:

- `stdin` — стандартный поток ввода (по умолчанию: клавиатура), файловый дескриптор 0;
- `stdout` — стандартный поток вывода (по умолчанию: консоль), файловый дескриптор 1;
- `stderr` — стандартный поток вывод сообщений об ошибках (по умолчанию: консоль), файловый дескриптор 2.

Большинство используемых в консоли команд и программ записывают результаты своей работы в стандартный поток вывода `stdout`. Например, команда `ls` выводит в стандартный поток вывода (консоль) список файлов в текущей директории. Потоки вывода и ввода можно перенаправлять на другие файлы или устройства. Проще всего это делается с помощью символов `>`, `>>`, `<`, `<<`. Рассмотрим пример.

```
Перенаправление stdout (вывода) в файл.
Если файл отсутствовал, то он создаётся,
иначе -- перезаписывается.

Создаёт файл, содержащий список дерева каталогов.
ls -lR > dir-tree.list

1>filename
Перенаправление вывода (stdout) в файл "filename".
1>>filename
Перенаправление вывода (stdout) в файл "filename",
файл открывается в режиме добавления.
2>filename
Перенаправление stderr в файл "filename".
2>>filename
Перенаправление stderr в файл "filename",
файл открывается в режиме добавления.
&>filename
Перенаправление stdout и stderr в файл "filename".
```

#### 4.2.2. Конвейер

Конвейер (`pipe`) служит для объединения простых команд или утилит в цепочки, в которых результат работы предыдущей команды передаётся последующей. Синтаксис следующий:

команда 1 | команда 2

# означает, что вывод команды 1 передастся на ввод команде 2

Конвейеры можно группировать в цепочки и выводить с помощью перенаправления в файл, например:

```
ls -la |sort > sortilg_list
```

вывод команды `ls -la` передаётся команде сортировки `sort\verb`, которая пишет результат в файл `sorting_list\verb`.

Чаще всего скрипты на Bash используются в качестве автоматизации каких-то рутинных операций в консоли, отсюда иногда возникает необходимость в обработке `stdout` одной команды и передача на `stdin` другой команде, при этом результат выполнения команды должен обработан.

### 4.2.3. Поиск файла

Команда `find` используется для поиска и отображения на экран имён файлов, соответствующих заданной строке символов.

Формат команды:

```
find путь [-опции]
```

Путь определяет каталог, начиная с которого по всем подкаталогам будет вестись поиск.

**Примеры:**

1. Вывести на экран имена файлов из вашего домашнего каталога и его подкаталогов, начинающихся на `f`:

```
find ~ -name "f*" -print
```

Здесь `~` — обозначение вашего домашнего каталога, `-name` — после этой опции указывается имя файла, который нужно найти, `"f*"` — строка символов, определяющая имя файла, `-print` — опция, задающая вывод результатов поиска на экран.

2. Вывести на экран имена файлов в каталоге `/etc`, начинающихся с символа `p`:

```
find /etc -name "p*" -print
```

3. Найти в Вашем домашнем каталоге файлы, имена которых заканчиваются символом `~` и удалить их:

```
find ~ -name "*~" -exec rm "{}" \;
```

Здесь опция `-exec rm "{}" \;` задаёт применение команды `rm` ко всем файлам, имена которых соответствуют указанной после опции `-name` строке символов.

Для просмотра опций команды `find` воспользуйтесь командой `man`.

### 4.2.4. Фильтрация текста

Найти в текстовом файле указанную строку символов позволяет команда `grep`.

Формат команды:

```
grep строка имя файла
```

Кроме того, команда `grep` способна обрабатывать стандартный вывод других команд (любой текст). Для этого следует использовать конвейер, связав вывод команды с вводом `grep`.

**Примеры:**

1. Показать строки во всех файлах в вашем домашнем каталоге с именами, начинающимися на `f`, в которых есть слово `begin`:

```
grep begin f*
```

2. Найти в текущем каталоге все файлы, содержащих в имени «лаб»:

```
ls -l | grep лаб
```

### 4.2.5. Проверка использования диска

Команда `df` показывает размер каждого смонтированного раздела диска.

Формат команды:

```
df [-опции] [файловая_система]
```

**Пример:**

```
df -vi
```

Команда `du` показывает число килобайт, используемое каждым файлом или каталогом.

Формат команды:

```
du [-опции] [имя_файла...]
```

**Пример.**

```
du -a ~/
```

На `afs` можно посмотреть использованное пространство командой

```
fs quota
```

### 4.2.6. Управление задачами

Любую выполняющуюся в консоли команду или внешнюю программу можно запустить в фоновом режиме. Для этого следует в конце имени команды указать знак амперсанда `&`. Например:

```
gedit &
```

Будет запущен текстовый редактор `gedit` в фоновом режиме. Консоль при этом не будет заблокирована.

Запущенные фоновыми программы называются задачами (`jobs`). Ими можно управлять с помощью команды `jobs`, которая выводит список запущенных в данный момент задач. Для завершения задачи необходимо выполнить команду

```
kill %номер_задачи
```

### 4.2.7. Управление процессами

Любой команде, выполняемой в системе, присваивается *идентификатор процесса* (*process ID*). Получить информацию о процессе и управлять им, пользуясь идентификатором процесса, можно из любого окна командного интерпретатора.

### 4.2.8. Получение информации о процессах

Команда `ps` используется для получения информации о процессах.

Формат команды:

```
ps [-опции]
```

Для получения информации о процессах, управляемых вами и запущенных (работающих или остановленных) на вашем терминале, используйте опцию `aux`.

**Пример:**

```
ps aux
```

Для запуска команды в фоновом режиме необходимо в конце командной строки указать знак `&` (амперсанд).

Пример работы, требующей много машинного времени для выполнения, и которую целесообразно запустить в фоновом режиме:

```
find /var/log -name "*.log" -print > 1.log &
```

### 4.3. Последовательность выполнения работы

1. Осуществите вход в систему, используя соответствующее имя пользователя.
2. Запишите в файл `file.txt` названия файлов, содержащихся в каталоге `/etc`. Допишите в этот же файл названия файлов, содержащихся в вашем домашнем каталоге.
3. Выведите имена всех файлов из `file.txt`, имеющих расширение `.conf`, после чего запишите их в новый текстовый файл `conf.txt`.
4. Определите, какие файлы в вашем домашнем каталоге имеют имена, начинавшиеся с символа `c`? Предложите несколько вариантов, как это сделать.
5. Выведите на экран (по странично) имена файлов из каталога `/etc`, начинающиеся с символа `h`.
6. Запустите в фоновом режиме процесс, который будет записывать в файл `~/logfile` файлы, имена которых начинаются с `log`.
7. Удалите файл `~/logfile`.
8. Запустите из консоли в фоновом режиме редактор `gedit`.
9. Определите идентификатор процесса `gedit`, используя команду `ps`, конвейер и фильтр `grep`. Как ещё можно определить идентификатор процесса?
10. Прочтите справку (`man`) команды `kill`, после чего используйте её для завершения процесса `gedit`.
11. Выполните команды `df` и `du`, предварительно получив более подробную информацию об этих командах, с помощью команды `man`.
12. Воспользовавшись справкой команды `find`, выведите имена всех директорий, имеющих в вашем домашнем каталоге.

### 4.4. Содержание отчёта

1. Титульный лист с указанием номера лабораторной работы и ФИО студента.
2. Формулировка цели работы.
3. Описание результатов выполнения задания:
  - скриншоты (снимки экрана), фиксирующие выполнение лаб. раб.;
  - листинги (исходный код) программ (если они есть);
  - результаты выполнения программ.
4. Выводы, согласованные с целью работы.
5. Ответы на контрольные вопросы.

### 4.5. Контрольные вопросы

1. Какие потоки ввода вывода вы знаете?
2. Объясните разницу между операцией `>` и `>>`.
3. Что такое конвейер?
4. Что такое процесс? Чем это понятие отличается от программы?
5. Что такое PID и GID?
6. Что такое задачи и какая команда позволяет ими управлять?
7. Найдите информацию об утилитах `top` и `htop`. Каковы их функции?
8. Назовите и дайте характеристику команде поиска файлов. Приведите примеры использования этой команды.
9. Можно ли по контексту (содержанию) найти файл? Если да, то как?
10. Как определить объем свободной памяти на жёстком диске?
11. Как определить объем вашего домашнего каталога?
12. Как удалить зависший процесс?

# Лабораторная работа № 5. Командная оболочка Midnight Commander

## 5.1. Цель работы

Освоение основных возможностей командной оболочки Midnight Commander. Приобретение навыков практической работы по просмотру каталогов и файлов; манипуляций с ними.

## 5.2. Указания к работе

### 5.2.1. Общие сведения

Командная оболочка — интерфейс взаимодействия пользователя с операционной системой и программным обеспечением посредством команд.

Midnight Commander (или mc) — псевдографическая командная оболочка для UNIX/Linux систем. Для запуска mc необходимо в командной строке набрать mc и нажать **Enter**.

Рабочее пространство mc имеет две панели, отображающие по умолчанию списки файлов двух каталогов (рис. 5.1).

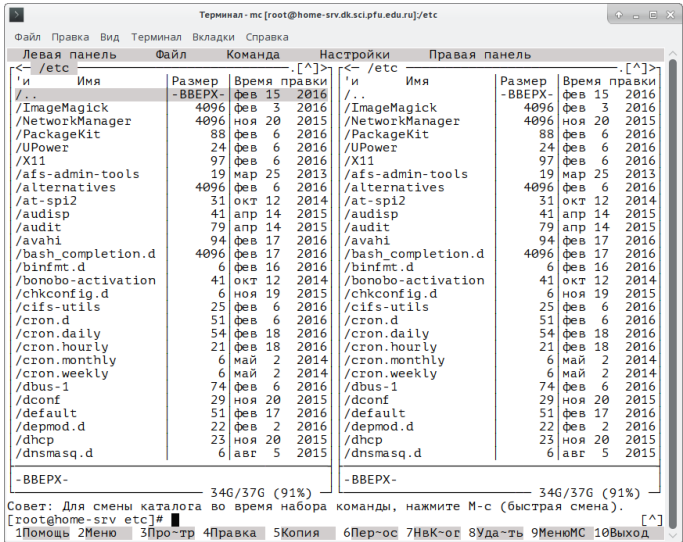


Рис. 5.1. Внешний вид экрана при работе с Midnight Commander

Над панелями располагается меню, доступ к которому осуществляется с помощью клавиши **F9**. Под панелями внизу расположены управляющие экранные кнопки

ки, ассоциированные с функциональными клавишами **F1**–**F10** (табл. 5.1). Над ними располагается командная строка, предназначенная для ввода команд.

Таблица 5.1

Функциональные клавиши **ms**

|            |                                                                                                                           |
|------------|---------------------------------------------------------------------------------------------------------------------------|
| <b>F1</b>  | Вызов контекстно-зависимой подсказки                                                                                      |
| <b>F2</b>  | Вызов пользовательского меню с возможностью создания и/или дополнения дополнительных функций                              |
| <b>F3</b>  | Просмотр содержимого файла, на который указывает подсветка в активной панели (без возможности редактирования)             |
| <b>F4</b>  | Вызов встроенного в <b>ms</b> редактора для изменения содержания файла, на который указывает подсветка в активной панели  |
| <b>F5</b>  | Копирование одного или нескольких файлов, отмеченных в первой (активной) панели, в каталог, отображаемый на второй панели |
| <b>F6</b>  | Перенос одного или нескольких файлов, отмеченных в первой (активной) панели, в каталог, отображаемый на второй панели     |
| <b>F7</b>  | Создание подкаталога в каталоге, отображаемом в активной панели                                                           |
| <b>F8</b>  | Удаление одного или нескольких файлов (каталогов), отмеченных в первой (активной) панели файлов                           |
| <b>F9</b>  | Вызов меню <b>ms</b>                                                                                                      |
| <b>F10</b> | Выход из <b>ms</b>                                                                                                        |

### 5.2.2. Режимы отображения панелей и управление ими

Панель в **ms** отображает список файлов текущего каталога. Абсолютный путь к этому каталогу отображается в заголовке панели. У активной панели заголовок и одна из её строк подсвечиваются. Управление панелями осуществляется с помощью определённых комбинаций клавиш или пунктов меню **ms**.

Панели можно поменять местами. Для этого и используется комбинация клавиш **Ctrl-u** или команда меню **ms** **Переставить панели**. Также можно временно убрать отображение панелей (отключить их) с помощью комбинации клавиш **Ctrl-o** или команды меню **ms** **Отключить панели**. Это может быть полезно, например, если необходимо увидеть вывод какой-то информации на экран после выполнения какой-либо команды **shell**.

С помощью последовательного применения комбинации клавиш **Ctrl-x** **d** есть возможность сравнения каталогов, отображённых на двух панелях. Панели могут дополнительно быть переведены в один из двух режимов: **Информация** или **Дерево**. В режиме **Информация** (рис. 5.2) на панель выводятся сведения о файле и текущей файловой системе, расположенных на активной панели. В режиме **Дерево** (рис. 5.3) на одной из панелей выводится структура дерева каталогов.

Управлять режимами отображения панелей можно через пункты меню **ms** **Правая панель** и **Левая панель** (рис. 5.4).

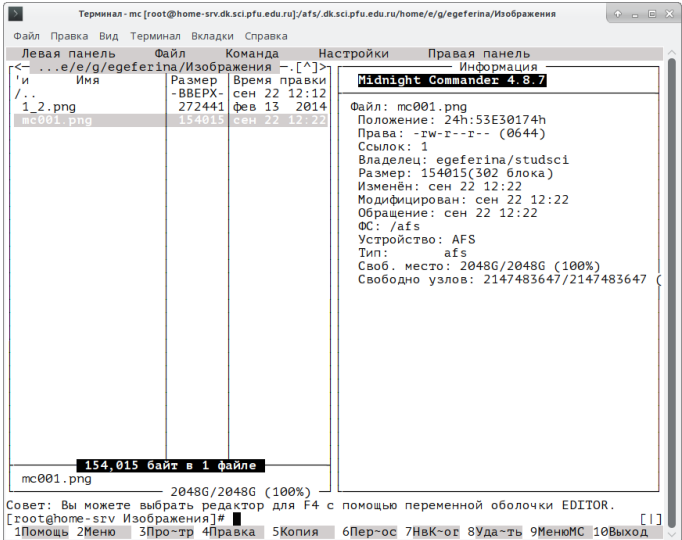


Рис. 5.2. Режим Информация

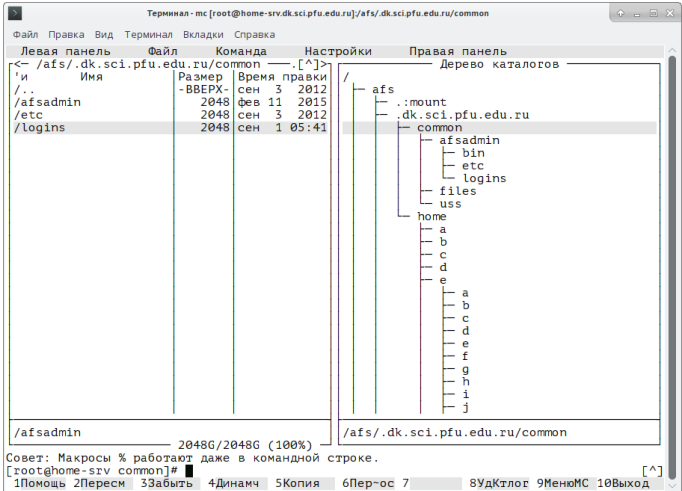


Рис. 5.3. Режим отображения дерева каталогов



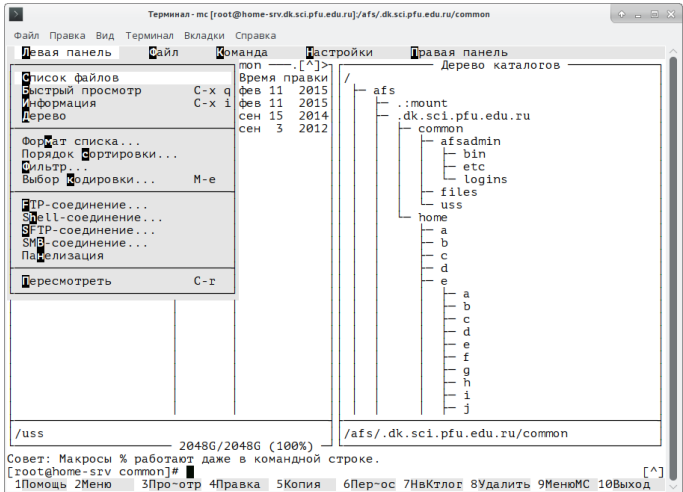


Рис. 5.4. Меню Левая Панель

5.2.3. Меню панелей

Перейти в строку меню панелей mc можно с помощью функциональной клавиши **F9**. В строке меню имеются пять меню: **Левая панель**, **Файл**, **Команда**, **Настройки** и **Правая панель**.

Подпункт меню **Быстрый просмотр** позволяет выполнить быстрый просмотр содержимого панели.

Подпункт меню **Информация** позволяет посмотреть информацию о файле или каталоге (рис. 5.5).

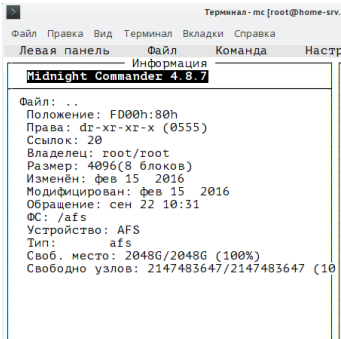


Рис. 5.5. Панель Информация

В меню каждой (левой или правой) панели можно выбрать **Формат списка**:

- стандартный — выводит список файлов и каталогов с указанием размера и времени правки;
- ускоренный — позволяет задать число столбцов, на которые разбивается панель при выводе списка имён файлов или каталогов без дополнительной информации;
- расширенный — помимо названия файла или каталога выводит сведения о правах доступа, владельцы, группе, размере, времени правки;
- определённый пользователем — позволяет вывести те сведения о файле или каталоге, которые задаст сам пользователь.

Подпункт меню **Порядок сортировки** позволяет задать критерии сортировки при выводе списка файлов и каталогов: без сортировки, по имени, расширенный, время правки, время доступа, время изменения атрибута, размер, узел.

### 5.2.3.1. Меню Файл

В меню **Файл** содержит перечень команд, которые могут быть применены к одному или нескольким файлам или каталогам (рис. 5.6).

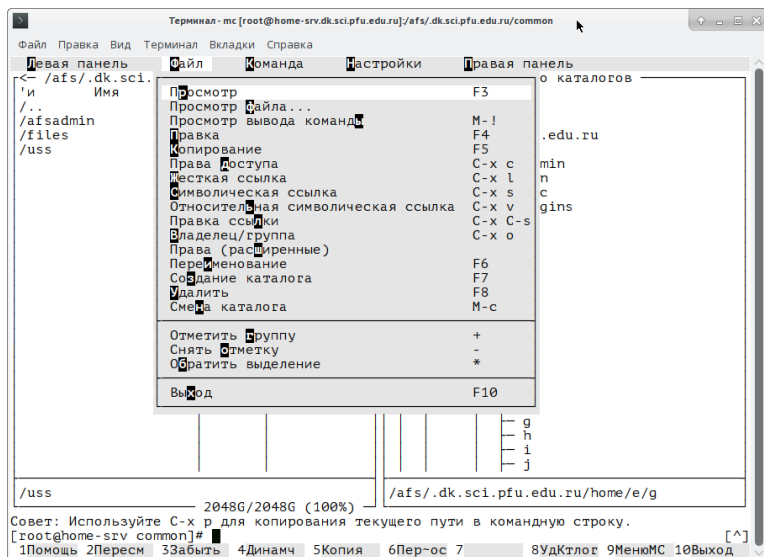


Рис. 5.6. Меню Файл

Команды меню **Файл**:

- Просмотр (**F3**) — позволяет посмотреть содержимое текущего (или выделенного) файла без возможности редактирования.
- Просмотр вывода команды (**M + !**) — функция запроса команды с параметрами (аргумент к текущему выбранному файлу).

- Правка (F4) — открывает текущий (или выделенный) файл для его редактирования.
- Копирование (F5) — осуществляет копирование одного или нескольких файлов или каталогов в указанное пользователем во всплывающем окне место.
- Права доступа (Ctrl-x c) — позволяет указать (изменить) права доступа к одному или нескольким файлам или каталогам (рис. 5.7).

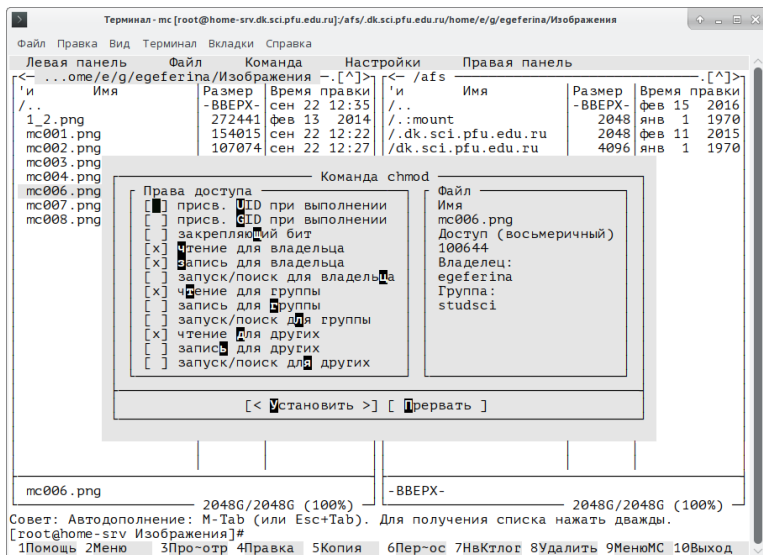


Рис. 5.7. Права доступа на файлы и каталоги

- Жёсткая ссылка (Ctrl-x l) — позволяет создать жёсткую ссылку к текущему (или выделенному) файлу<sup>1</sup>.
- Символическая ссылка (Ctrl-x s) — позволяет создать символическую ссылку к текущему (или выделенному) файлу<sup>2</sup>.
- Владелец/группа (Ctrl-x o) — позволяет задать (изменить) владельца и имя группы для одного или нескольких файлов или каталогов.
- Права (расширенные) — позволяет изменить права доступа и владения для одного или нескольких файлов или каталогов.
- Переименование (F6) — позволяет переименовать (или переместить) один или несколько файлов или каталогов.
- Создание каталога (F7) — позволяет создать каталог.
- Удалить (F8) — позволяет удалить один или несколько файлов или каталогов.
- Выход (F10) — завершает работу mc.

<sup>1</sup> Жёсткая ссылка проявляется как реальный файл. После её создания невозможно определить, где сам файл, а где ссылка на него. Если удалить один из этих файлов, то другой останется целым.

<sup>2</sup> Символическая ссылка — ссылка (указатель) на имя файла-оригинала.

### 5.2.3.2. Меню Команда

В меню **Команда** содержатся более общие команды для работы с mc (рис. 5.8).

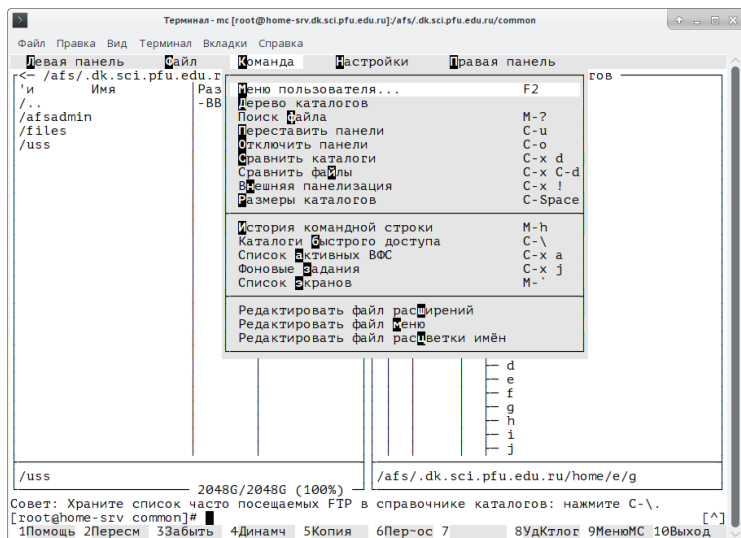


Рис. 5.8. Меню Команда

Команды меню **Команда**:

- Дерево каталогов — отображает структуру каталогов системы.
- Поиск файла — выполняет поиск файлов по заданным параметрам.
- Переставить панели — меняет местами левую и правую панели.
- Сравнить каталоги (**Ctrl-x d**) — сравнивает содержимое двух каталогов.
- Размеры каталогов — отображает размер и время изменения каталога (по умолчанию в mc размер каталога корректно не отображается).
- История командной строки — выводит на экран список ранее выполненных в оболочке команд.
- Каталоги быстрого доступа (**Ctrl-\**) — при вызове выполняется быстрая смена текущего каталога на один из заданного списка.
- Восстановление файлов — позволяет восстановить файлы на файловых системах ext2 и ext3.
- Редактировать файл расширений — позволяет задать с помощью определённого синтаксиса действия при запуске файлов с определённым расширением (например, какое программное обеспечение запускать для открытия или редактирования файлов с расширением doc или docx).
- Редактировать файл меню — позволяет отредактировать контекстное меню пользователя, вызываемое по клавише **F2**.
- Редактировать файл расцветки имён — позволяет подобрать оптимальную для пользователя расцветку имён файлов в зависимости от их типа.

### 5.2.3.3. Меню Настройки

Меню **Настройки** содержит ряд дополнительных опций по внешнему виду и функциональности **mc** (рис. 5.9).

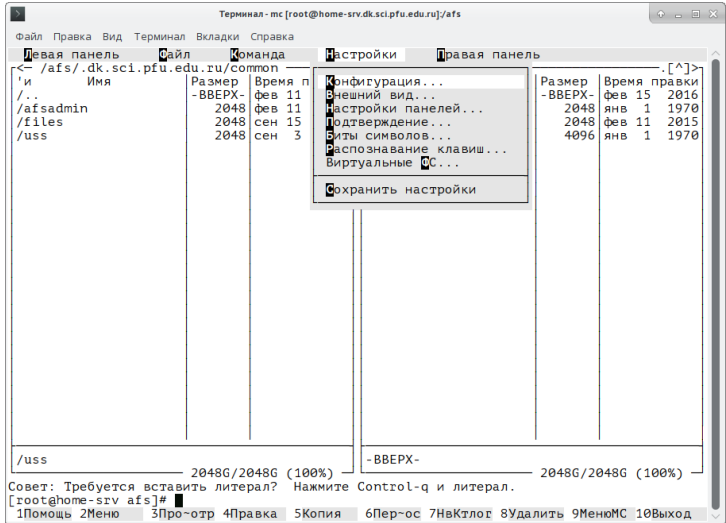


Рис. 5.9. Меню Настройки

- Меню **Настройки** содержит:
- **Конфигурация** — позволяет скорректировать настройки работы с панелями.
  - **Внешний вид и Настройки панелей** — определяет элементы (строка меню, командная строка, подсказки и прочее), отображаемые при вызове **mc**, а также геометрию расположения панелей и цветовыделение.
  - **Биты символов** — задаёт формат обработки информации локальным терминалом.
  - **Подтверждение** — позволяет установить или убрать вывод окна с запросом подтверждения действий при операциях удаления и перезаписи файлов, а также при выходе из программы.
  - **Распознавание клавиш** — диалоговое окно используется для тестирования функциональных клавиш, клавиш управления курсором и прочее.
  - **Виртуальные ФС** — настройки виртуальной файловой системы: тайм-аут, пароль и прочее.

### 5.2.4. Редактор **mc**

Встроенный в **mc** редактор вызывается с помощью функциональной клавиши **F4**. В нём удобно использовать различные комбинации клавиш при редактировании содержимого (как правило текстового) файла (табл. 5.2).

Таблица 5.2

## Клавиши для редактирования файла

|        |                                                                 |
|--------|-----------------------------------------------------------------|
| Ctrl-y | удалить строку                                                  |
| Ctrl-u | отмена последней операции                                       |
| Ins    | вставка/замена                                                  |
| F7     | поиск (можно использовать регулярные выражения)                 |
| ⇧-F7   | повтор последней операции поиска                                |
| F4     | замена                                                          |
| F3     | первое нажатие — начало выделения, второе — окончание выделения |
| F5     | копировать выделенный фрагмент                                  |
| F6     | переместить выделенный фрагмент                                 |
| F8     | удалить выделенный фрагмент                                     |
| F2     | записать изменения в файл                                       |
| F10    | выйти из редактора                                              |

## 5.3. Последовательность выполнения работы

## 5.3.1. Задание по mc

1. Изучите информацию о mc, вызвав в командной строке `man mc`.
2. Запустите из командной строки mc, изучите его структуру и меню.
3. Выполните несколько операций в mc, используя управляющие клавиши (операции с панелями; выделение/отмена выделения файлов, копирование/перемещение файлов, получение информации о размере и правах доступа на файлы и/или каталоги и т.п.)
4. Выполните основные команды меню левой (или правой) панели. Оцените степень подробности вывода информации о файлах.
5. Используя возможности подменю **Файл**, выполните:
  - просмотр содержимого текстового файла;
  - редактирование содержимого текстового файла (без сохранения результатов редактирования);
  - создание каталога;
  - копирование в файлов в созданный каталог.
6. С помощью соответствующих средств подменю **Команда** осуществите:
  - поиск в файловой системе файла с заданными условиями (например, файла с расширением `.c` или `.cpp`, содержащего строку `main`);
  - выбор и повторение одной из предыдущих команд;
  - переход в домашний каталог;
  - анализ файла меню и файла расширений.
7. Вызовите подменю **Настройки**. Освойте операции, определяющие структуру экрана mc (Full screen, Double Width, Show Hidden Files и т.д.)

### 5.3.2. Задание по встроенному редактору `mc`

1. Создайте текстовый файл `text.txt`.
2. Откройте этот файл с помощью встроенного в `mc` редактора.
3. Вставьте в открытый файл небольшой фрагмент текста, скопированный из любого другого файла или Интернета.
4. Прodelайте с текстом следующие манипуляции, используя горячие клавиши:
  - 4.1. Удалите строку текста.
  - 4.2. Выделите фрагмент текста и скопируйте его на новую строку.
  - 4.3. Выделите фрагмент текста и перенесите его на новую строку.
  - 4.4. Сохраните файл.
  - 4.5. Отмените последнее действие.
  - 4.6. Перейдите в конец файла (нажав комбинацию клавиш) и напишите некоторый текст.
  - 4.7. Перейдите в начало файла (нажав комбинацию клавиш) и напишите некоторый текст.
  - 4.8. Сохраните и закройте файл.
5. Откройте файл с исходным текстом на некотором языке программирования (например `C` или `Java`)
6. Используя меню редактора, включите подсветку синтаксиса, если она не включена, или выключите, если она включена.

### 5.4. Содержание отчёта

1. Титульный лист с указанием номера лабораторной работы и ФИО студента.
2. Формулировка цели работы.
3. Описание результатов выполнения задания:
  - скриншоты (снимки экрана), фиксирующие выполнение лабораторной работы;
  - листинги (исходный код) программ (если они есть);
  - результаты выполнения программ (текст или снимок экрана в зависимости от задания).
4. Выводы, согласованные с целью работы.
5. Ответы на контрольные вопросы.

### 5.5. Контрольные вопросы

1. Какие режимы работы есть в `mc`. Охарактеризуйте их.
2. Какие операции с файлами можно выполнить как с помощью команд `shell`, так и с помощью меню (комбинаций клавиш) `mc`? Приведите несколько примеров.
3. Опишите структура меню левой (или правой) панели `mc`, дайте характеристику командам.
4. Опишите структура меню **Файл** `mc`, дайте характеристику командам.
5. Опишите структура меню **Команда** `mc`, дайте характеристику командам.
6. Опишите структура меню **Настройки** `mc`, дайте характеристику командам.
7. Назовите и дайте характеристику встроенным командам `mc`.
8. Назовите и дайте характеристику командам встроенного редактора `mc`.
9. Дайте характеристику средствам `mc`, которые позволяют создавать меню, определяемые пользователем.
10. Дайте характеристику средствам `mc`, которые позволяют выполнять действия, определяемые пользователем, над текущим файлом.

# Лабораторная работа № 6. Текстовый редактор vi

## 6.1. Цель работы

Познакомиться с операционной системой Linux. Получить практические навыки работы с редактором vi, установленным по умолчанию практически во всех дистрибутивах.

## 6.2. Указания к работе

В большинстве дистрибутивов Linux в качестве текстового редактора по умолчанию устанавливается интерактивный экранный редактор vi (Visual display editor).

- Редактор vi имеет три режима работы:
- *командный режим* — предназначен для ввода команд редактирования и навигации по редактируемому файлу;
  - *режим вставки* — предназначен для ввода содержания редактируемого файла;
  - *режим последней (или командной) строки* — используется для записи изменений в файл и выхода из редактора.

Для вызова редактора vi необходимо указать команду vi и имя редактируемого файла:

```
vi <имя_файла>
```

При этом в случае отсутствия файла с указанным именем будет создан такой файл.

Переход в командный режим осуществляется нажатием клавиши **[Esc]**. Для выхода из редактора vi необходимо перейти в режим последней строки: находясь в командном режиме, нажать **[Shift;]** (по сути символ : — двоеточие), затем:

- набрать символы wq, если перед выходом из редактора требуется записать изменения в файл;
- набрать символ q (или q!), если требуется выйти из редактора без сохранения.

Замечание. Следует помнить, что vi различает прописные и строчные буквы при наборе (восприятии) команд.

### 6.2.1. Основные группы команд редактора

#### 6.2.1.1. Команды управления курсором

Команды управления курсором приведены в табл. 6.1.

Таблица 6.1

Команды управления курсором

| Курсор влево                                | Курсор вправо                                  | Курсор вверх              | Курсор вниз               |
|---------------------------------------------|------------------------------------------------|---------------------------|---------------------------|
| <div>←</div> <div>(клавиша Backspace)</div> | <div>Space</div> <div>(клавиша «пробел»)</div> |                           | <div>Enter</div>          |
| <div>h</div> <div>←</div>                   | <div>l</div> <div>→</div>                      | <div>k</div> <div>↑</div> | <div>j</div> <div>↓</div> |



### 6.2.1.2. Команды позиционирования

- (ноль) — переход в начало строки;
- — переход в конец строки;
- — переход в конец файла;
- $n$  — переход на строку с номером  $n$ .

### 6.2.1.3. Команды перемещения по файлу

- — перейти на пол-экрана вперёд;
- — перейти на пол-экрана назад;
- — перейти на страницу вперёд;
- — перейти на страницу назад.

### 6.2.1.4. Команды перемещения по словам<sup>1</sup>

- или — перейти на слово вперёд;
- $n$  или  $n$  — перейти на  $n$  слов вперёд;
- или — перейти на слово назад;
- $n$  или  $n$  — перейти на  $n$  слов назад.

## 6.2.2. Команды редактирования

### 6.2.2.1. Вставка текста

- — вставить текст после курсора;
- — вставить текст в конец строки;
- — вставить текст перед курсором;
- $n$  — вставить текст  $n$  раз;
- — вставить текст в начало строки.

### 6.2.2.2. Вставка строки





- — вставить строку под курсором;
- — вставить строку над курсором.

### 6.2.2.3. Удаление текста



- — удалить один символ в буфер;
- — удалить одно слово в буфер;
- — удалить в буфер текст от курсора до конца строки;
- — удалить в буфер текст от начала строки до позиции курсора;

---





<sup>1</sup> При использовании прописных W и B под разделителями понимаются только пробел, табуляция и возврат каретки. При использовании строчных w и b под разделителями понимаются также любые знаки пунктуации.

-   — удалить в буфер одну строку;
- $n$    — удалить в буфер  $n$  строк.



#### 6.2.2.4. Отмена и повтор произведённых изменений

-  — отменить последнее изменение;
-  — повторить последнее изменение.









#### 6.2.2.5. Копирование текста в буфер

-  — скопировать строку в буфер;
- $n$   — скопировать  $n$  строк в буфер;
-   — скопировать слово в буфер.



#### 6.2.2.6. Вставка текста из буфера

-  — вставить текст из буфера после курсора;
-  — вставить текст из буфера перед курсором.

#### 6.2.2.7. Замена текста









-   — заменить слово;
- $n$    — заменить  $n$  слов;
-   — заменить текст от курсора до конца строки;
-  — заменить слово;
-  — заменить текст.

#### 6.2.2.8. Поиск текста



















-  *текст* — произвести поиск вперёд по тексту указанной строки символов *текст*;
-  *текст* — произвести поиск назад по тексту указанной строки символов *текст*.

### 6.2.3. Команды редактирования в режиме командной строки

#### 6.2.3.1. Копирование и перемещение текста





-   $n, m$   — удалить строки с  $n$  по  $m$ ;
-   $i, j$    $k$  — переместить строки с  $i$  по  $j$ , начиная со строки  $k$ ;
-   $i, j$    $k$  — копировать строки с  $i$  по  $j$  в строку  $k$ ;
-   $i, j$   *имя-файла* — записать строки с  $i$  по  $j$  в файл с именем *имя-файла*.

### 6.2.3.2. Запись в файл и выход из редактора

-   **w** — записать изменённый текст в файл, не выходя из vi;
-   **w** *имя-файла* — записать изменённый текст в новый файл с именем *имя-файла*;
-   **w**  **!** *имя-файла* — записать изменённый текст в файл с именем *имя-файла*;
-   **w**  **q** — записать изменения в файл и выйти из vi;
-   **q** — выйти из редактора vi;
-   **q**  **!** — выйти из редактора без записи;
-   **e**  **!** — вернуться в командный режим, отменив все изменения, произведённые со времени последней записи.

### 6.2.4. Опции

Опции редактора vi позволяют настроить рабочую среду. Для задания опций используется команда `set` (в режиме последней строки):


-  `set all` — вывести полный список опций;
-  `set nu` — вывести номера строк;
-  `set list` — вывести невидимые символы;
-  `set ic` — не учитывать при поиске, является ли символ прописным или строчным.


Если вы хотите отказаться от использования опции, то в команде `set` перед именем опции надо поставить `no`.

## 6.3. Последовательность выполнения работы

1. Ознакомиться с теоретическим материалом.
2. Ознакомиться с редактором vi.
3. Выполнить упражнения, используя команды vi.

### 6.3.1. Задание 1. Создание нового файла с использованием vi

1. Создайте каталог с именем `~/work/os/lab06`.
2. Перейдите во вновь созданный каталог.
3. Вызовите vi и создайте файл `hello.sh`  
`vi hello.sh`
4. Нажмите клавишу  **i** и вводите следующий текст.  

```
#!/bin/bash
HELL=Hello
function hello {
 LOCAL HELLO=World
 echo $HELLO
}
echo $HELLO
hello
```
5. Нажмите клавишу  **Esc** для перехода в командный режим после завершения ввода текста.

6. Нажмите **:** для перехода в режим последней строки и внизу вашего экрана появится приглашение в виде двоеточия.
7. Нажмите **w** (записать) и **q** (выйти), а затем нажмите клавишу **Enter** для сохранения вашего текста и завершения работы.
8. Сделайте файл исполняемым  
`chmod +x hello.sh`

### 6.3.2. Задание 2. Редактирование существующего файла

1. Вызовите vi на редактирование файла  
`vi ~/work/os/lab06/hello.sh`
2. Установите курсор в конец слова `HELL` второй строки.
3. Перейдите в режим вставки и замените на `HELLO`. Нажмите **Esc** для возврата в командный режим.
4. Установите курсор на четвертую строку и сотрите слово `LOCAL`.
5. Перейдите в режим вставки и наберите следующий текст: `local`, нажмите **Esc** для возврата в командный режим.
6. Установите курсор на последней строке файла. Вставьте после неё строку, содержащую следующий текст: `echo $HELLO`.
7. Нажмите **Esc** для перехода в командный режим.
8. Удалите последнюю строку.
9. Введите команду отмены изменений **u** для отмены последней команды.
10. Введите символ **:** для перехода в режим последней строки. Запишите произведённые изменения и выйдите из vi.

### 6.4. Содержание отчёта

1. Титульный лист с указанием номера лабораторной работы и ФИО студента.
2. Формулировка цели работы.
3. Описание результатов выполнения задания:
  - скриншоты (снимки экрана), фиксирующие выполнение лабораторной работы;
  - листинги (исходный код) программ (если они есть);
  - результаты выполнения программ (текст или снимок экрана в зависимости от задания).
4. Выводы, согласованные с целью работы.
5. Ответы на контрольные вопросы.

### 6.5. Контрольные вопросы

1. Дайте краткую характеристику режимам работы редактора vi.
2. Как выйти из редактора, не сохраняя произведённые изменения?
3. Назовите и дайте краткую характеристику командам позиционирования.
4. Что для редактора vi является словом?
5. Каким образом из любого места редактируемого файла перейти в начало (конец) файла?
6. Назовите и дайте краткую характеристику основным группам команд редактирования.
7. Необходимо заполнить строку символами \$. Каковы ваши действия?

8. Как отменить некорректное действие, связанное с процессом редактирования?
9. Назовите и дайте характеристику основным группам команд режима последней строки.
10. Как определить, не перемещая курсора, позицию, в которой заканчивается строка?
11. Выполните анализ опций редактора vi (сколько их, как узнать их назначение и т.д.).
12. Как определить режим работы редактора vi?
13. Постройте граф взаимосвязи режимов работы редактора vi.

## Лабораторная работа № 7. Текстовый редактор `emacs`

### 7.1. Цель работы

Познакомиться с операционной системой Linux. Получить практические навыки работы с редактором Emacs.

### 7.2. Указания к работе

Emacs представляет собой мощный экраный редактор текста, написанный на языке высокого уровня Elisp.

#### 7.2.1. Основные термины Emacs

Определение 8. Буфер — объект, представляющий какой-либо текст.

Буфер может содержать что угодно, например, результаты компиляции программы или встроенные подсказки. Практически всё взаимодействие с пользователем, в том числе интерактивное, происходит посредством буферов.

Определение 9. Фрейм соответствует окну в обычном понимании этого слова. Каждый фрейм содержит область вывода и одно или несколько окон Emacs.

Определение 10. Окно — прямоугольная область фрейма, отображающая один из буферов.

Каждое окно имеет свою строку состояния, в которой выводится следующая информация: название буфера, его основной режим, изменялся ли текст буфера и как далеко вниз по буферу расположен курсор. Каждый буфер находится только в одном из возможных основных режимов. Существующие основные режимы включают режим `Fundamental` (наименее специализированный), режим `Text`, режим `Lisp`, режим `C`, режим `Texinfo` и другие. Под второстепенными режимами понимается список режимов, которые включены в данный момент в буфере выбранного окна.

Определение 11. *Область вывода* — одна или несколько строк внизу фрейма, в которой Emacs выводит различные сообщения, а также запрашивает подтверждения и дополнительную информацию от пользователя.

Определение 12. *Минибуфер* используется для ввода дополнительной информации и всегда отображается в области вывода.

Определение 13. *Точка вставки* — место вставки (удаления) данных в буфере.

#### 7.2.2. Основы работы в Emacs

Для запуска Emacs необходимо в командной строке набрать `emacs` (или `emacs &` для работы в фоновом режиме относительно консоли).

Для работы с Emacs можно использовать как элементы меню, так и различные сочетания клавиш. Например, для выхода из Emacs можно воспользоваться меню

**File** и выбрать пункт **Quit**, а можно нажать последовательно **Ctrl-x** **Ctrl-c** (в обозначениях Emacs: **C-x** **C-c**).

Многие рутинные операции в Emacs удобнее производить с помощью клавиатуры, а не графического меню. Наиболее часто в командах Emacs используются сочетания с клавишами **Ctrl** и **Meta** (в обозначениях Emacs: **C-** и **M-**; клавиша **Shift** в Emacs обозначается как **S-**). Так как на клавиатуре для IBM PC совместимых ПК клавиша **Meta** нет, то вместо неё можно использовать **Alt** или **Esc**. Для доступа к системе меню используйте клавишу **F10**.

Клавиши **Ctrl**, **Meta** и **Shift** принято называть префиксными. Например, запись **M-x** означает, что надо удерживая клавишу **Meta** (или **Alt**), нажать на клавишу **x**. Для открытия файла следует использовать команду **C-x** **C-f** (надо, удерживая клавишу **Ctrl**, нажать на клавишу **x**, затем отпустить обе клавиши и снова, удерживая клавишу **Ctrl**, нажать на клавишу **f**).

По назначению префиксные сочетания клавиш различаются следующим образом:

- **C-x** — префикс ввода основных команд редактора (например, открытия, закрытия, сохранения файла и т.д.);
- **C-c** — префикс вызова функций, зависящих от используемого режима.

**Определение 14.** *Режим* — пакет расширений, изменяющий поведение буфера Emacs при редактировании и просмотре текста (например, для редактирования исходного текста программ на языках C или Perl).

В табл. 7.1 приведены основные комбинации клавиш, используемые для перемещения курсора в буфере Emacs (также работают и обычные навигационные клавиши, например, стрелки).

**Таблица 7.1**  
**Основные комбинации клавиш для перемещения курсора в буфере Emacs**

| Комбинация клавиш | Действие                             |
|-------------------|--------------------------------------|
| <b>C-p</b>        | переместиться вверх на одну строку   |
| <b>C-n</b>        | переместиться вниз на одну строку    |
| <b>C-f</b>        | переместиться вперёд на один символ  |
| <b>C-b</b>        | переместиться назад на один символ   |
| <b>C-a</b>        | переместиться в начало строки        |
| <b>C-e</b>        | переместиться в конец строки         |
| <b>C-v</b>        | переместиться вниз на одну страницу  |
| <b>M-v</b>        | переместиться вверх на одну страницу |
| <b>M-f</b>        | переместиться вперёд на одно слово   |
| <b>M-b</b>        | переместиться назад на одно слово    |
| <b>M-&lt;</b>     | переместиться в начало буфера        |
| <b>M-&gt;</b>     | переместиться в конец буфера         |
| <b>C-g</b>        | закончить текущую операцию           |

Далее в табл. 7.2–7.7 приведены наиболее часто используемые комбинации клавиш для выполнения действий в Emacs.

Таблица 7.2

**Основные комбинации клавиш для работы с текстом в Emacs**

| Комбинация клавиш | Действие                                                                |
|-------------------|-------------------------------------------------------------------------|
| C-d               | Удалить символ перед текущим положением курсора                         |
| M-d               | Удалить следующее за текущим положением курсора слово                   |
| C-k               | Удалить текст от текущего положения курсора до конца строки             |
| M-k               | Удалить текст от текущего положения курсора до конца предложения        |
| M-\               | Удалить все пробелы и знаки табуляции вокруг текущего положения курсора |
| C-q               | Вставить символ, соответствующий нажатой клавише или сочетанию          |
| M-q               | Выводить текст в текущем параграфе буфера                               |

Таблица 7.3

**Основные комбинации клавиш для работы с выделенной областью текста в Emacs**

| Комбинация клавиш | Действие                                                    |
|-------------------|-------------------------------------------------------------|
| C-space           | Начать выделение текста с текущего положения курсора        |
| C-w               | Удалить выделенную область текста в список удалений         |
| M-w               | Скопировать выделенную область текста в список удалений     |
| C-y               | Вставить текст из списка удалений в текущую позицию курсора |
| M-y               | Последовательно вставить текст из списка удалений           |
| M-\               | Выводить строки выделенной области текста                   |

Таблица 7.4

**Основные комбинации клавиш для поиска и замены в Emacs**

| Комбинация клавиш | Действие                                                   |
|-------------------|------------------------------------------------------------|
| C-s текст поиска  | Поиск текста в прямом направлении                          |
| C-r текст поиска  | Поиск текста в обратном направлении                        |
| M-%               | Поиск текста и его замена с запросом (что на что заменить) |



Таблица 7.5

**Основные комбинации клавиш для работы с файлами, буферами и окнами в Emacs**

| Комбинация клавиш | Действие                                                    |
|-------------------|-------------------------------------------------------------|
| C-x C-f           | Открыть файл                                                |
| C-x C-s           | Сохранить текст в буфер                                     |
| C-x C-b           | Отобразить список открытых буферов в новом окне             |
| C-x b             | Переключиться в другой буфер в текущем окне                 |
| C-x i             | Вставить содержимое файла в буфер в текущую позицию курсора |
| C-x 0             | Закрыть текущее окно (при этом буфер не удаляется)          |
| C-x 1             | Закрыть все окна кроме текущего                             |
| C-x 2             | Разделить окно по горизонтали                               |
| C-x o             | Перейти в другое окно                                       |

Таблица 7.6

**Основные комбинации клавиш для работы со справкой в Emacs**

| Комбинация клавиш | Действие                                                                 |
|-------------------|--------------------------------------------------------------------------|
| C-h ?             | Показать информацию по работе со справочной системой                     |
| C-h t             | Вызвать интерактивный учебник                                            |
| C-h f             | Показать информацию по функции                                           |
| C-h v             | Показать информацию по переменной                                        |
| C-h k             | Показать информацию по действию комбинации клавиш                        |
| C-h a             | Выполнить поисковый запрос в справке по строке или регулярному выражению |
| C-h F             | Вызвать Emacs FAQ                                                        |
| C-h i             | Показать документацию по Emacs (Info)                                    |

Таблица 7.7

**Прочие комбинации клавиш, используемые в Emacs**

| Комбинация клавиш | Действие                                 |
|-------------------|------------------------------------------|
| C-\               | Переключить язык                         |
| M-x command       | Выполнить команду Emacs с именем command |
| C-x u             | Отменить последнюю операцию              |





### 7.2.3. Регулярные выражения

При работе с командами Emacs можно использовать регулярные выражения (табл. 7.8). Основные отличия от PCRE (Perl Compatible Regular Expressions — библиотека регулярных выражений в стиле Perl):

- `\s` не задаёт пробел;
- `\t` не задаёт табуляцию;
- операция «или» и скобки группировки экранируются.

Таблица 7.8



Регулярные выражения в Emacs

|                                                                                   |                                                                                   |                                                      |
|-----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|------------------------------------------------------|
|  |  | новая строка                                         |
|  |  | табуляция                                            |
| .                                                                                 |                                                                                   | любой знак кроме новой строки                        |
| *                                                                                 |                                                                                   | повторение предыдущего 0–n раз, жадное               |
| +                                                                                 |                                                                                   | повторение предыдущего 1–n раз, жадное               |
| ?                                                                                 |                                                                                   | повторение предыдущего 0–1 раз, жадное               |
| *?, +?, ??                                                                        |                                                                                   | аналогично предыдущим, ленивые                       |
| [ ... ]                                                                           |                                                                                   | набор символов, ^ в начале строки — «не эти символы» |
| ^                                                                                 |                                                                                   | начало строки                                        |
| \$                                                                                |                                                                                   | конец строки                                         |
| \                                                                                 |                                                                                   | или                                                  |
| \(...\)                                                                           |                                                                                   | группировка                                          |
| \b                                                                                |                                                                                   | граница слова                                        |
| \B                                                                                |                                                                                   | не граница слова                                     |
| \w                                                                                |                                                                                   | буквенный символ                                     |
| \W                                                                                |                                                                                   | небуквенный символ                                   |
| \1                                                                                |                                                                                   | ссылка на первую группу                              |

## 7.3. Последовательность выполнения работы

1. Ознакомиться с теоретическим материалом.
2. Ознакомиться с редактором `emacs`.
3. Выполнить упражнения.
4. Ответить на контрольные вопросы.

### 7.3.1. Основные команды `emacs`

1. Открыть `emacs`.
2. Создать файл `lab07.sh` с помощью комбинации   (`C-x C-f`).
3. Наберите текст:
 

```
#!/bin/bash
HELL=Hello
function hello {
```

```

LOCAL HELLO=World
echo $HELLO
}
echo $HELLO
hello

```

4. Сохранить файл с помощью комбинации **Ctrl-x Ctrl-s** (C-x C-s).
5. Прodelать с текстом стандартные процедуры редактирования, каждое действие должно осуществляться комбинацией клавиш.
  - 5.1. Вырезать одной командой целую строку (C-k).
  - 5.2. Вставить эту строку в конец файла (C-y).
  - 5.3. Выделить область текста (C-space).
  - 5.4. Скопировать область в буфер обмена (M-w).
  - 5.5. Вставить область в конец файла.
  - 5.6. Вновь выделить эту область и на этот раз вырезать её (C-w).
  - 5.7. Отмените последнее действие (C-/).
6. Научитесь использовать команды по перемещению курсора.
  - 6.1. Переместите курсор в начало строки (C-a).
  - 6.2. Переместите курсор в конец строки (C-e).
  - 6.3. Переместите курсор в начало буфера (M-<).
  - 6.4. Переместите курсор в конец буфера (M->).
7. Управление буферами.
  - 7.1. Вывести список активных буферов на экран (C-x C-b).
  - 7.2. Переместитесь во вновь открытое окно (C-x) о со списком открытых буферов и переключитесь на другой буфер.
  - 7.3. Закройте это окно (C-x 0).
  - 7.4. Теперь вновь переключайтесь между буферами, но уже без вывода их списка на экран (C-x b).
8. Управление окнами.
  - 8.1. Поделите фрейм на 4 части: разделите фрейм на два окна по вертикали (C-x 3), а затем каждое из этих окон на две части по горизонтали (C-x 2) (см. рис. 7.1).

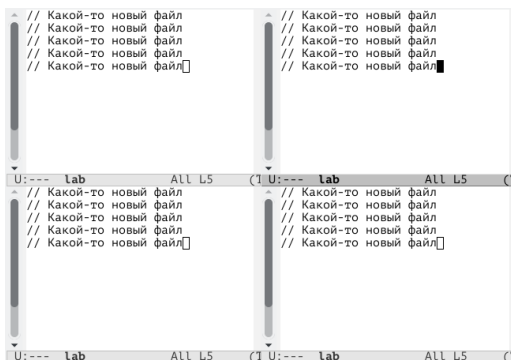


Рис. 7.1. Фрейм emacs, разделённый на 4 окна

- 8.2. В каждом из четырёх созданных окон откройте новый буфер (файл) и введите несколько строк текста.

## 9. Режим поиска

- 9.1. Переключитесь в режим поиска (`C-s`) и найдите несколько слов, присутствующих в тексте.
- 9.2. Переключайтесь между результатами поиска, нажимая `C-s`.
- 9.3. Выйдите из режима поиска, нажав `C-g`.
- 9.4. Перейдите в режим поиска и замены (`M-%`), введите текст, который следует найти и заменить, нажмите `[Enter]`, затем введите текст для замены. После того как будут подсвечены результаты поиска, нажмите `!` для подтверждения замены.
- 9.5. Испробуйте другой режим поиска, нажав `M-s` о. Объясните, чем он отличается от обычного режима?

## 7.4. Содержание отчёта

1. Титульный лист с указанием номера лабораторной работы и ФИО студента.
2. Формулировка цели работы.
3. Описание результатов выполнения задания:
  - скриншоты (снимки экрана), фиксирующие выполнение лабораторной работы;
  - листинги (исходный код) программ (если они есть);
  - результаты выполнения программ (текст или снимок экрана в зависимости от задания).
4. Выводы, согласованные с целью работы.
5. Ответы на контрольные вопросы.

## 7.5. Контрольные вопросы

1. Кратко охарактеризуйте редактор `emacs`.
2. Какие особенности данного редактора могут сделать его сложным для освоения новичком?
3. Своими словами опишите, что такое буфер и окно в терминологии `emacs`'а.
4. Можно ли открыть больше 10 буферов в одном окне?
5. Какие буферы создаются по умолчанию при запуске `emacs`?
6. Какие клавиши вы нажмёте, чтобы ввести следующую комбинацию `C-c |` и `C-c C-|`?
7. Как поделить текущее окно на две части?
8. В каком файле хранятся настройки редактора `emacs`?
9. Какую функцию выполняет клавиша `[←]` и можно ли её переназначить?
10. Какой редактор вам показался удобнее в работе `vi` или `emacs`? Поясните почему.

## Лабораторная работа № 8. Программирование в командном процессоре ОС UNIX. Командные файлы

### 8.1. Цель работы

Изучить основы программирования в оболочке ОС UNIX/Linux. Научиться писать небольшие командные файлы.

### 8.2. Указания к лабораторной работе

#### 8.2.1. Командные процессоры (оболочки)

*Командный процессор (командная оболочка, интерпретатор команд shell)* — это программа, позволяющая пользователю взаимодействовать с операционной системой компьютера. В операционных системах типа UNIX/Linux наиболее часто используются следующие реализации командных оболочек:

- *оболочка Борна (Bourne shell или sh)* — стандартная командная оболочка UNIX/Linux, содержащая базовый, но при этом полный набор функций;
- *C-оболочка (или csh)* — надстройка на оболочке Борна, использующая C-подобный синтаксис команд с возможностью сохранения истории выполнения команд;
- *оболочка Корна (или ksh)* — напоминает оболочку C, но операторы управления программой совместимы с операторами оболочки Борна;
- *BASH* — сокращение от Bourne Again Shell (опять оболочка Борна), в основе своей совмещает свойства оболочек C и Корна (разработка компании Free Software Foundation).

*POSIX (Portable Operating System Interface for Computer Environments)* — набор стандартов описания интерфейсов взаимодействия операционной системы и прикладных программ.

Стандарты POSIX разработаны комитетом IEEE (Institute of Electrical and Electronics Engineers) для обеспечения совместимости различных UNIX/Linux-подобных операционных систем и переносимости прикладных программ на уровне исходного кода. POSIX-совместимые оболочки разработаны на базе оболочки Корна.

Рассмотрим основные элементы программирования в оболочке bash. В других оболочках большинство команд будет совпадать с описанными ниже.

#### 8.2.2. Переменные в языке программирования bash

Командный процессор bash обеспечивает возможность использования переменных типа *строка символов*. Имена переменных могут быть выбраны пользователем. Пользователь имеет возможность присвоить переменной значение некоторой строки символов. Например, команда

```
mark=/usr/andy/bin
```

присваивает значение строки символов /usr/andy/bin переменной mark типа *строка символов*.

Значение, присвоенное некоторой переменной, может быть впоследствии использовано. Для этого в соответствующем месте командной строки должно быть употреблено имя этой переменной, которому предшествует метасимвол \$. Например, команда

`mv afile ${mark}`  
 переместит файл `afile` из текущего каталога в каталог с абсолютным полным именем `/usr/andy/bin`.

Использование значения, присвоенного некоторой переменной, называется подстановкой. Для того чтобы имя переменной не сливалось с символами, которые могут следовать за ним в командной строке, при подстановке в общем случае используется следующая форма записи:

`${имя переменной}`

Например, использование команд

`b=/tmp/andy-`

`ls -l myfile > ${b}ls`  
`sudo apt-get install texlive-luatex`

приведёт к переназначению стандартного вывода команды `ls` с терминала на файл `/tmp/andy-ls`, а использование команды `ls -l>${b}ls` приведёт к подстановке в командную строку значения переменной `ls`. Если переменной `ls` не было предварительно присвоено никакого значения, то её значением будет символ пробела.

Оболочка `bash` позволяет работать с массивами. Для создания массива используется команда `set` с флагом `-A`. За флагом следует имя переменной, а затем список значений, разделённых пробелами. Например,

`set -A states Delaware Michigan "New Jersey"`

Далее можно сделать добавление в массив, например, `states[49]=Alaska`. Индексация массивов начинается с нулевого элемента.

### 8.2.3. Использование арифметических вычислений. Операторы `let` и `read`

Оболочка `bash` поддерживает встроенные арифметические функции. Команда `let` является показателем того, что последующие аргументы представляют собой выражение, подлежащее вычислению. Простейшее выражение — это единичный терм (`term`), обычно целочисленный.

Целые числа можно записывать как последовательность цифр или в любом базовом формате типа `radix\#number`, где `radix` (основание системы счисления) — любое число не более 26. Для большинства команд используются следующие основания систем исчисления: 2 (двоичная), 8 (восьмеричная) и 16 (шестнадцатеричная). Простейшими математическими выражениями являются сложение (+), вычитание (-), умножение (\*), целочисленное деление (/) и целочисленный остаток от деления (%).

Команда `let` берет два операнда и присваивает их переменной. Положительным моментом команды `let` можно считать то, что для идентификации переменной ей не нужен знак доллара; вы можете писать команды типа `let sum=x+7`, и `let` будет искать переменную `x` и добавлять к ней 7.

Команда `let` также расширяет другие выражения `let`, если они заключены в двойные круглые скобки. Таким способом вы можете создавать довольно сложные выражения.

Команда `let` не ограничена простыми арифметическими выражениями. Табл. 8.1 показывает полный набор `let`-операций.

Подобно `C` оболочка `bash` может присваивать переменной любое значение, а произвольное выражение само имеет значение, которое может использоваться. При этом «ноль» воспринимается как «ложь», а любое другое значение выражения — как «истина». Для облегчения программирования можно записывать условия оболочки `bash` в двойные скобки — `(( ))`.

Таблица 8.1

Арифметические операторы оболочки **bash**

| Оператор | Синтаксис    | Результат                                                                         |
|----------|--------------|-----------------------------------------------------------------------------------|
| !        | !exp         | Если exp равно 0, то возвращает 1; иначе 0                                        |
| !=       | exp1 !=exp2  | Если exp1 не равно exp2, то возвращает 1; иначе 0                                 |
| %        | exp1%exp2    | Возвращает остаток от деления exp1 на exp2                                        |
| %=       | var=%exp     | Присваивает остаток от деления var на exp переменной var                          |
| &        | exp1&exp2    | Возвращает побитовое AND выражений exp1 и exp2                                    |
| &&       | exp1&&exp2   | Если и exp1 и exp2 не равны нулю, то возвращает 1; иначе 0                        |
| &=       | var &= exp   | Присваивает переменной var побитовое AND var и exp                                |
| *        | exp1 * exp2  | Умножает exp1 на exp2                                                             |
| *=       | var *= exp   | Умножает exp на значение переменной var и присваивает результат переменной var    |
| +        | exp1 + exp2  | Складывает exp1 и exp2                                                            |
| +=       | var += exp   | Складывает exp со значением переменной var и результат присваивает переменной var |
| -        | -exp         | Операция отрицания exp (унарный минус)                                            |
| -        | exp1 - exp2  | Вычитает exp2 из exp1                                                             |
| -=       | var -= exp   | Вычитает exp из значения переменной var и присваивает результат переменной var    |
| /        | exp / exp2   | Делит exp1 на exp2                                                                |
| /=       | var /= exp   | Делит значение переменной var на exp и присваивает результат переменной var       |
| <        | exp1 < exp2  | Если exp1 меньше, чем exp2, то возвращает 1, иначе возвращает 0                   |
| <<       | exp1 << exp2 | Сдвигает exp1 влево на exp2 бит                                                   |
| <<=      | var <<= exp  | Побитовый сдвиг влево значения переменной var на exp                              |
| <=       | exp1 <= exp2 | Если exp1 меньше или равно exp2, то возвращает 1; иначе возвращает 0              |
| =        | var = exp    | Присваивает значение exp переменной var                                           |
| ==       | exp1==exp2   | Если exp1 равно exp2, то возвращает 1; иначе возвращает 0                         |
| >        | exp1 > exp2  | 1, если exp1 больше, чем exp2; иначе 0                                            |
| >=       | exp1 >= exp2 | 1, если exp1 больше или равно exp2; иначе 0                                       |
| >>       | exp >> exp2  | Сдвигает exp1 вправо на exp2 бит                                                  |
| >>=      | var >>=exp   | Побитовый сдвиг вправо значения переменной var на exp                             |
| ^        | exp1 ^ exp2  | Исключающее OR выражений exp1 и exp2                                              |
| ^=       | var ^= exp   | Присваивает переменной var побитовое XOR var и exp                                |
|          | exp1   exp2  | Побитовое OR выражений exp1 и exp2                                                |
| =        | var  = exp   | Присваивает переменной var результат операции XOR var и exp                       |
|          | exp1    exp2 | 1, если или exp1 или exp2 являются ненулевыми значениями; иначе 0                 |
| ~        | ~exp         | Побитовое дополнение до exp                                                       |

Можно присваивать результаты условных выражений переменным, также как и использовать результаты арифметических вычислений в качестве условий. Хорошим примером сказанного является выполнение некоторого действия, одновременно декрементируя некоторое значение. например:

```
$ let x=5
$ while
> ((x-=1))
> do
> something
> done
```

Этот пример показывает выполнение некоторого действия с начальным значением 5, которое декрементирует до тех пор, пока оно не будет равно нулю. При каждой итерации выполняется функция `something`.

Наиболее распространённым является сокращение, избавляющееся от слова `let` в программах оболочек. Если объявить переменные целыми значениями, то любое присвоение автоматически будет трактоваться как арифметическое действие. Если использовать `typeset -i` для объявления и присвоения переменной, то при последующем её применении она станет целой. Также можно использовать ключевое слово `integer` (псевдоним для `typeset -i`) и объявлять таким образом переменные целыми. Выражения типа `x=y+z` будет восприниматься в это случае как арифметические.

Команда `read` позволяет читать значения переменных со стандартного ввода:

```
echo "Please enter Month and Day of Birth ?"
read mon day trash
```

В переменные `mon` и `day` будут считаны соответствующие значения, введённые с клавиатуры, а переменная `trash` нужна для того, чтобы отобрать всю избыточно введённую информацию и игнорировать её.

Изъять переменную из программы можно с помощью команды `unset`.

Имена некоторых переменных имеют для командного процессора специальный смысл. Значением переменной `PATH` (т.е. `$PATH`) является список каталогов, в которых командный процессор осуществляет поиск программы или команды, указанной в командной строке, в том случае, если указанное имя программы или команды не содержит ни одного символа `/`. Если имя команды содержит хотя бы один символ `/`, то последовательность поиска, предписываемая значением переменной `PATH`, нарушается. В этом случае в зависимости от того, является имя команды абсолютным или относительно, поиск начинается соответственно от корневого или текущего каталога.

Если Вы сами явно не присвоите переменной `PATH` какое-либо значение, то стандартной (по умолчанию) последовательностью поиска файла является следующая: текущий каталог, каталог `/bin`, каталог `/usr/bin`. Именно в такой последовательности командный процессор ищет файлы, содержащие программы, которые обеспечивают выполнение таких, например, команд, как `echo`, `ls` и `cat`.

В списке каталогов, являющемся значением переменной `PATH`, имена каталогов отделяются друг от друга с помощью символа двоеточия. В качестве примера приведём команду:

```
PATH=~:/bin:/usr/local/bin/:/bin:/usr/bin
```

Переменные `PS1` и `PS2` предназначены для отображения промптера командного процессора. `PS1` — это промптер командного процессора, по умолчанию его значение равно символу `$` или `#`. Если какая-то интерактивная программа, запущенная командным процессором, требует ввода, то используется промптер `PS2`. Он по умолчанию имеет значение символа `>`.

Другие стандартные переменные:



- HOME — имя домашнего каталога пользователя. Если команда `cd` вводится без аргументов, то происходит переход в каталог, указанный в этой переменной.
- IFS — последовательность символов, являющихся разделителями в командной строке, например, пробел, табуляция и перевод строки (new line).
- MAIL — командный процессор каждый раз перед выводом на экран промптера проверяет содержимое файла, имя которого указано в этой переменной, и если содержимое этого файла изменилось с момента последнего ввода из него, то перед тем как вывести на терминал промптер, командный процессор выводит на терминал сообщение `You have mail` (у Вас есть почта).
- TERM — тип используемого терминала.
- LOGNAME — содержит регистрационное имя пользователя, которое устанавливается автоматически при входе в систему.

В командном процессоре Си имеется ещё несколько стандартных переменных. Значение всех переменных можно просмотреть с помощью команды `set`.

## 8.2.4. Метасимволы и их экранирование

При перечислении имён файлов текущего каталога можно использовать следующие символы:

- `*` — соответствует произвольной, в том числе и пустой строке;
- `?` — соответствует любому одинарному символу;
- `[c1-c1]` — соответствует любому символу, лексикографически находящемуся между символами `c1` и `c2`.

Например,

- `echo *` — выведет имена всех файлов текущего каталога, что представляет собой простейший аналог команды `ls`;
- `ls *.c` — выведет все файлы с последними двумя символами, совпадающими с `.c`.
- `echo prog.?` — выведет все файлы, состоящие из пяти или шести символов, первыми пятью символами которых являются `prog.`
- `[a-z]*` — соответствует произвольному имени файла в текущем каталоге, начинающемуся с любой строчной буквы латинского алфавита.

Такие символы, как `' < > * ? | \ " &`, являются метасимволами и имеют для командного процессора специальный смысл. Снятие специального смысла с метасимвола называется *экранированием метасимвола*. Экранирование может быть осуществлено с помощью предшествующего метасимволу символа `\`, который, в свою очередь, является метасимволом.

Для экранирования группы метасимволов нужно заключить её в одинарные кавычки. Строка, заключённая в двойные кавычки, экранирует все метасимволы, кроме `$, ' , \, "`. Например,

- `echo \*` выведет на экран символ `*`,
- `echo ab'\*|\*'cd` выведет на экран строку `ab*\|*cd`.

## 8.2.5. Командные файлы и функции

Последовательность команд может быть помещена в текстовый файл. Такой файл называется *командным*. Далее этот файл можно выполнить по команде:

```
bash командный_файл [аргументы]
```

Чтобы не вводить каждый раз последовательности символов `bash`, необходимо изменить код защиты этого командного файла, обеспечив доступ к этому файлу по выполнению. Это может быть сделано с помощью команды

```
chmod +x имя_файла
```

Теперь можно вызывать свой командный файл на выполнение, просто вводя его имя с терминала так, как-будто он является выполняемой программой. Командный процессор распознает, что в Вашем файле на самом деле хранится не выполняемая программа, а программа, написанная на языке программирования оболочки, и осуществит её интерпретацию.

Группу команд можно объединить в функцию. Для этого существует ключевое слово `function`, после которого следует имя функции и список команд, заключённых в фигурные скобки. Удалить функцию можно с помощью команды `unset` с флагом `-f`.

Команда `typeset` имеет четыре опции для работы с функциями:

- `-f` — перечисляет определённые на текущий момент функции;
- `-ft` — при последующем вызове функции иницирует её трассировку;
- `-fx` — экспортирует все перечисленные функции в любые дочерние программы оболочек;
- `-fu` — обозначает указанные функции как автоматически загружаемые. Автоматически загружаемые функции хранятся в командных файлах, а при их вызове оболочка просматривает переменную `FRATH`, отыскивая файл с одноимёнными именами функций, загружает его и вызывает эти функции.

## 8.2.6. Передача параметров в командные файлы и специальные переменные

При вызове командного файла на выполнение параметры ему могут быть переданы точно таким же образом, как и выполняемой программе. С точки зрения командного файла эти параметры являются позиционными. Символ `$` является метасимволом командного процессора. Он используется, в частности, для ссылки на параметры, точнее, для получения их значений в командном файле. В командный файл можно передать до девяти параметров. При использовании где-либо в командном файле комбинации символов `$i`, где  $0 < i < 10$ , вместо неё будет осуществлена подстановка значения параметра с порядковым номером  $i$ , т.е. аргумента командного файла с порядковым номером  $i$ . Использование комбинации символов `$0` приводит к подстановке вместо неё имени данного командного файла. Рассмотрим это на примере.

Пусть к командному файлу `where` имеется доступ по выполнению и этот командный файл содержит следующий конвейер:

```
who | grep $1.
```

Если Вы введёте с терминала команду `where andy`, то в случае, если пользователь, зарегистрированный в ОС UNIX под именем `andy`, в данный момент работает в ОС UNIX, то на терминал будет выведена строка, содержащая номер терминала, используемого указанным пользователем. Если же в данный момент этот пользователь не работает в ОС UNIX, то на терминал ничего не будет выведено.

Команда `grep` производит контекстный поиск в тексте, поступающем со стандартного ввода, для нахождения в этом тексте строк, содержащих последовательности символов, переданные ей в качестве аргументов, и выводит результаты своей работы на стандартный вывод. В примере команда `grep` используется как фильтр, обеспечивающий ввод со стандартного ввода и вывод всех строк, содержащих последовательность символов `andy`, на стандартный вывод.

В ходе интерпретации файла командным процессором вместо комбинации символов `$1` осуществляется подстановка значения первого и единственного параметра `andy`. Если предположить, что пользователь, зарегистрированный в ОС UNIX под именем `andy`, в данный момент работает в ОС UNIX, то на терминале Вы увидите примерно следующее:

```
$ where andy
andy ttyG Jan 14 09:12
$
```

Определим функцию, которая изменяет каталог и печатает список файлов:

```
$ function clist {
> cd $1
> ls
> }
```

Теперь при вызове команды `clist` будет изменён каталог и выведено его содержимое.

Команда `shift` позволяет удалять первый параметр и сдвигает все остальные на места предыдущих.

При использовании в командном файле комбинации символов `$#` вместо неё будет осуществлена подстановка числа параметров, указанных в командной строке при вызове данного командного файла на выполнение.

Вот ещё несколько специальных переменных, используемых в командных файлах:

- `$*` — отображается вся командная строка или параметры оболочки;
- `$?` — код завершения последней выполненной команды;
- `$$_` — уникальный идентификатор процесса, в рамках которого выполняется командный процессор;
- `$!` — номер процесса, в рамках которого выполняется последняя вызванная на выполнение в командном режиме команда;
- `$-` — значение флагов командного процессора;
- `${#*}` — возвращает целое число — количество слов, которые были результатом `$*`;
- `${#name}` — возвращает целое значение длины строки в переменной `name`;
- `${name[n]}` — обращение к `n`-му элементу массива;
- `${name[*]}` — перечисляет все элементы массива, разделённые пробелом;
- `${name[@]}` — то же самое, но позволяет учитывать символы пробелы в самих переменных;
- `${name:-value}` — если значение переменной `name` не определено, то оно будет заменено на указанное `value`;
- `${name:value}` — проверяется факт существования переменной;
- `${name=value}` — если `name` не определено, то ему присваивается значение `value`;
- `${name?value}` — останавливает выполнение, если имя переменной не определено, и выводит `value` как сообщение об ошибке;
- `${name+value}` — это выражение работает противоположно `${name-value}`. Если переменная определена, то подставляется `value`;
- `${name#pattern}` — представляет значение переменной `name` с удалённым самым коротким левым образцом (`pattern`);
- `${#name[*]}` и `${#name[@]}` — эти выражения возвращают количество элементов в массиве `name`.

## 8.2.7. Использование команды `getopts`

Весьма необходимой при программировании является команда `getopts`, которая осуществляет синтаксический анализ командной строки, выделяя флаги, и используется для объявления переменных. Синтаксис команды следующий:

```
getopts option-string variable [arg ...]
```

**Флаги** — это опции командной строки, обычно помеченные знаком минус; Например, для команды `ls` флагом может являться `-f`. Иногда флаги имеют аргументы, связанные с ними. Программы интерпретируют флаги, соответствующим образом изменяя своё поведение.

Строка опций `option-string` — это список возможных букв и чисел соответствующего флага. Если ожидается, что некоторый флаг будет сопровождаться некоторым аргументом, то за символом, обозначающим этот флаг, должно следовать двоеточие. Соответствующей переменной присваивается буква данной опции. Если команда `getopts` может распознать аргумент, то она возвращает истину. Принято включать `getopts` в цикл `while` и анализировать введённые данные с помощью оператора `case`.

Предположим, необходимо распознать командную строку следующего формата:

```
testprog -ifile_in.txt -ofile_out.doc -L -t -r
```

Вот как выглядит использование оператора `getopts` в этом случае:

```
while getopts o:i:Ltr optletter
do case $optletter in
o) oflag=1; oval=$OPTARG;;
i) iflag=1; ival=$OPTARG;;
L) Lflag=1;;
t) tflag=1;;
r) rflag=1;;
*) echo Illegal option $optletter
esac
done
```

Функция `getopts` включает две специальные переменные среды — `OPTARG` и `OPTIND`. Если ожидается дополнительное значение, то `OPTARG` устанавливается в значение этого аргумента (будет равно `file_in.txt` для опции `i` и `file_out.doc` для опции `o`. `OPTIND` является числовым индексом на упомянутый аргумент.

Функция `getopts` также понимает переменные типа массив, следовательно, можно использовать её в функции не только для синтаксического анализа аргументов функций, но и для анализа введённых пользователем данных.

### 8.2.8. Управление последовательностью действий в командных файлах

Часто бывает необходимо обеспечить проведение каких-либо действий циклически и управление дальнейшими действиями в зависимости от результатов проверки некоторого условия. Для решения подобных задач язык программирования `bash` предоставляет возможность использовать такие управляющие конструкции, как `for`, `case`, `if` и `while`. С точки зрения командного процессора эти управляющие конструкции являются обычными командами и могут использоваться как при создании командных файлов, так и при работе в интерактивном режиме. Команды, реализующие подобные конструкции, по сути, являются операторами языка программирования `bash`. Поэтому при описании языка программирования `bash` термин *оператор* будет использоваться наравне с термином *команда*.

Команды ОС UNIX возвращают код завершения, значение которого может быть использовано для принятия решения о дальнейших действиях. Команда `test`, например, создана специально для использования в командных файлах. Единственная функция этой команды заключается в выработке кода завершения. Так например, команда

```
test -f file
```

возвращает нулевой код завершения (*истина*), если файл `file` существует, и ненулевой код завершения (*ложь*) в противном случае:

- `test s` — истина, если аргумент `s` имеет значение *истина*;
- `test -f file` — истина, если файл `file` существует;
- `test -i file` — истина, если файл `file` доступен по чтению;
- `test -w file` — истина, если файл `file` доступен по записи;
- `test -e file` — истина, если файл `file` — исполняемая программа;
- `test -d file` — истина, если файл `file` является каталогом.

### 8.2.8.1. Оператор цикла `for`

В обобщённой форме оператор цикла `for` выглядит следующим образом:

```
for имя [in список-значений]
do список-команд
done
```

При каждом следующем выполнении оператора цикла `for` переменная *имя* принимает следующее значение из списка значений, задаваемых списком *список-значений*. Вообще говоря, *список-значений* является необязательным. При его отсутствии оператор цикла `for` выполняется для всех позиционных параметров или, иначе говоря, аргументов. Таким образом, оператор `for i` эквивалентен оператору `for i in $*`. Выполнение оператора цикла `for` завершается, когда *список-значений* будет исчерпан. Последовательность команд (операторов), задаваемая списком *список-команд*, состоит из одной или более команд оболочки, отделённых друг от друга с помощью символов `newline` или `;`.

Рассмотрим примеры использования оператора цикла `for`.

В результате выполнения оператора

```
for A in alpha beta gamma
do echo A
done
```

на терминал будет выведено следующее:

```
alpha
beta
gamma
```

Предположим, что Вы хотите найти во всех файлах текущего каталога, содержащих исходные тексты программ, написанных на языке программирования Си, все вхождения функции с некоторым именем. Это можно сделать с помощью такой последовательности команд:

```
for i
do
 grep $i *.c
done
```

Поместив эту последовательность команд в файл `findref`, после возможно, используя команду

```
findref 'hash(' 'insert(' 'symbol(' ,
вывести на терминал все строки из всех файлов текущего каталога, имена которых оканчиваются символами .c, содержащие ссылки на функции hash(), insert() и symbol(). Использование символов ' в вышеприведённом примере необходимо для снятия специального смысла с символа (.
```

### 8.2.8.2. Оператор выбора case

Оператор выбора `case` реализует возможность ветвления на произвольное число ветвей. Эта возможность обеспечивается в большинстве современных языков программирования, предполагающих использование структурного подхода.

В обобщённой форме оператор выбора `case` выглядит следующим образом:

```
case имя in
шаблон1) список-команд;;
шаблон2) список-команд;;
...
esac
```

Выполнение оператора выбора `case` сводится к тому, что выполняется последовательность команд (операторов), задаваемая списком *список-команд*, в строке, для которой значение переменной *имя* совпадает с *шаблоном*. Поскольку метасимвол `*` соответствует произвольной, в том числе и пустой, последовательности символов, то его можно использовать в качестве *шаблона* в последней строке перед служебным словом `esac`. В этом случае реализуются все действия, которые необходимо произвести, если значение переменной *имя* не совпадает ни с одним из шаблонов, заданных в предшествующих строках.

Рассмотрим примеры использования оператора выбора `case`.

В результате выполнения оператора

```
for A in alpha beta gamma
 do case $A in
 alpha) B=a;;
 beta) B=c;;
 gamma) B=e
 esac
 echo $B
done
```

на терминал будет выведено следующее:

```
a
c
e
```

### 8.2.8.3. Условный оператор if

В обобщённой форме условный оператор `if` выглядит следующим образом:

```
if список-команд
then список-команд
{elif список-команд
then список-команд}
[else список-команд]
fi
```

Выполнение условного оператора `if` сводится к тому, что сначала выполняется последовательность команд (операторов), которую задаёт *список-команд* в строке, содержащей служебное слово `if`. Затем, если последняя выполненная команда из этой последовательности команд возвращает нулевой код завершения (*истина*), то будет выполнена последовательность команд (операторов), которую задаёт *список-команд* в строке, содержащей служебное слово `then`. Фраза `elif` проверяется в том случае, когда предыдущая проверка была ложной. Строка, содержащая служебное слово `else`, является необязательной. Если она присутствует, то последовательность команд (операторов), которую задаёт *список-команд* в строке, содержащей служебное слово `else`, будет выполнена только при условии, что последняя

выполненная команда из последовательности команд (операторов), которую задаёт *список-команд* в строке, содержащей служебное слово `if` или `elif`, возвращает ненулевой код завершения (*ложь*).

Рассмотрим следующий пример:

```
for A in *
do if test -d $A
then echo $A: is a directory
else echo -n $A: is a file and
 if test -w $A
 then echo writeable
 elif test -r $A
 then echo readable
 else echo neither readable nor writeable
fi
fi
done
```

Первая строка в приведённом выше примере обеспечивает выполнение всех последующих действий в цикле для всех имён файлов из текущего каталога. При этом переменная `A` на каждом шаге последовательно принимает значения, равные именам этих файлов. Первая содержащая служебное слово `if` строка проверяет, является ли файл, имя которого представляет собой текущее значение переменной `A`, каталогом. Если этот файл является каталогом, то на стандартный вывод выводятся имя этого файла и сообщение о том, что файл с указанным именем является каталогом. Эти действия в приведённом выше примере обеспечиваются в результате выполнения третьей строки.

Оставшиеся строки выполняются только в том случае, если проверка того, является ли файл, имя которого представляет собой текущее значение переменной `A`, каталогом, даёт отрицательный ответ. Это означает, что файл, имя которого представляет собой текущее значение переменной `A`, является обычным файлом. Если этот файл является обычным файлом, то на стандартный вывод выводятся имя этого файла и сообщение о том, что файл с указанным именем является обычным файлом. Эти действия в приведённом выше примере обеспечиваются в результате выполнения четвёртой строки. Особенностью использования команды `echo` в этой строке является использование флага `-n`, благодаря чему выводимая командой `echo` строка не будет дополнена символом `newline` (перевод строки), что позволяет впоследствии дополнить эту строку, как это, например, показано в приведённом выше примере.

Вторая строка, содержащая служебное слово `if`, проверяет, доступен ли по записи файл, имя которого представляет собой текущее значение переменной `A`. Если этот файл доступен по записи, то строка дополняется соответствующим сообщением. Если же этот файл недоступен по записи, то проверяется, доступен ли этот файл по чтению. Эти действия в приведённом выше примере обеспечиваются в результате выполнения седьмой строки. Если этот файл доступен по чтению, то строка дополняется соответствующим сообщением. Если же этот файл недоступен ни по записи, ни по чтению, то строка также дополняется соответствующим сообщением. Эти действия в приведённом выше примере обеспечиваются в результате выполнения девятой строки.

#### 8.2.8.4. Операторы цикла `while` и `until`

В обобщённой форме оператор цикла `while` выглядит следующим образом:

```

while список-команд
do список-команд
done

```

Выполнение оператора цикла `while` сводится к тому, что сначала выполняется последовательность команд (операторов), которую задаёт *список-команд* в строке, содержащей служебное слово `while`, а затем, если последняя выполненная команда из этой последовательности команд возвращает нулевой код завершения (*истина*), выполняется последовательность команд (операторов), которую задаёт *список-команд* в строке, содержащей служебное слово `do`, после чего осуществляется безусловный переход на начало оператора цикла `while`. Выход из цикла будет осуществлён тогда, когда последняя выполненная команда из последовательности команд (операторов), которую задаёт *список-команд* в строке, содержащей служебное слово `while`, возвратит ненулевой код завершения (*ложь*).

Приведённый ниже фрагмент командного файла иллюстрирует использование оператора цикла `while`. В нем реализуется ожидание события, состоящего в удалении файла с определённым именем, и только после наступления этого события производятся дальнейшие действия:

```

while test -f lockfile
do sleep 30
 echo waiting for semaphore
done

```

```
:create the semaphore file
```

```
echo > lockfile
```

```
:further commands and after them delete the semaphore file
```

```
rm lockfile
```

Командный файл, продемонстрированный в приведённом примере, по сути, является простейшей реализацией механизма синхронизации взаимодействующих процессов на основе семафоров.

При замене в операторе цикла `while` служебного слова `while` на `until` условие, при выполнении которого осуществляется выход из цикла, меняется на противоположное. В остальном оператор цикла `while` и оператор цикла `until` идентичны.

В обобщённой форме оператор цикла `until` выглядит следующим образом:

```

until список-команд
do список-команд
done

```

Следующие две команды ОС UNIX используются только совместно с управляющими конструкциями языка программирования `bash`: это команда `true`, которая всегда возвращает код завершения, равный нулю (т.е. *истина*), и команда `false`, которая всегда возвращает код завершения, не равный нулю (т.е. *ложь*).

Ниже приведены два примера, иллюстрирующие бесконечные циклы, которые будут выполняться до тех пор, пока ЭВМ не сломается или не будет выключена (ну, по крайней мере, до тех пор, пока Вы не нажмёте клавишу, соответствующую специальному символу `INTERRUPT`):

```

while true
do echo hello andy
done

```



```
until false
do echo hello mike
done
```

### 8.2.8.5. Прерывание циклов

Два несложных способа позволяют вам прерывать циклы в оболочке `bash`. Команда `break` завершает выполнение цикла, а команда `continue` завершает данную итерацию блока операторов.

Команда `break` полезна для завершения цикла `while` в ситуациях, когда условие перестает быть правильным. Пример бесконечного цикла `while` с прерыванием в момент, когда файл перестаёт существовать:

```
while true
do
 if [! -f $file]
 then
 break
 fi
 sleep 10
done
```

Команда `continue` используется в ситуациях, когда больше нет необходимости выполнять блок операторов, но вы можете захотеть продолжить проверять данный блок на других условиях выражениях. Пример предназначен для игнорирования файла `/dev/null` в произвольном списке:

```
while file=$filelist[$i]
 (($i < ${#filelist[*]}))
do
 if
 ["$file" == "dev/null"]
 then
 continue
 fi
 action
done
```

Эта программа пропускает нужное значение, но продолжает тестировать остальные.

## 8.3. Последовательность выполнения работы

1. Написать скрипт, который при запуске будет делать резервную копию самого себя (то есть файла, в котором содержится его исходный код) в другую директорию `backup` в вашем домашнем каталоге. При этом файл должен архивироваться одним из архиваторов на выбор `zip`, `bzip2` или `tar`. Способ использования команд архивации необходимо узнать, изучив справку.
2. Написать пример командного файла, обрабатывающего любое произвольное число аргументов командной строки, в том числе *превышающее* десять. Например, скрипт может последовательно распечатывать значения всех переданных аргументов.
3. Написать командный файл — аналог команды `ls` (без использования самой этой команды и команды `dir`). Требуется, чтобы он выдавал информацию о нужном каталоге и выводил информацию о возможностях доступа к файлам этого каталога.

4. Написать командный файл, который получает в качестве аргумента командной строки формат файла (.txt, .doc, .jpg, .pdf и т.д.) и вычисляет количество таких файлов в указанной директории. Путь к директории также передаётся в виде аргумента командной строки.

## 8.4. Содержание отчёта

1. Титульный лист с указанием номера лабораторной работы и ФИО студента.
2. Формулировка цели работы.
3. Описание результатов выполнения задания:
  - скриншоты (снимки экрана), фиксирующие выполнение лабораторной работы;
  - листинги (исходный код) программ (если они есть);
  - результаты выполнения программ (текст или снимок экрана в зависимости от задания).
4. Выводы, согласованные с целью работы.
5. Ответы на контрольные вопросы.

## 8.5. Контрольные вопросы

1. Объясните понятие командной оболочки. Приведите примеры командных оболочек. Чем они отличаются?
2. Что такое POSIX?
3. Как определяются переменные и массивы в языке программирования bash?
4. Каково назначение операторов `let` и `read`?
5. Какие арифметические операции можно применять в языке программирования bash?
6. Что означает операция `(( ))`?
7. Какие стандартные имена переменных Вам известны?
8. Что такое метасимволы?
9. Как экранировать метасимволы?
10. Как создавать и запускать командные файлы?
11. Как определяются функции в языке программирования bash?
12. Каким образом можно выяснить, является файл каталогом или обычным файлом?
13. Каково назначение команд `set`, `typeset` и `unset`?
14. Как передаются параметры в командные файлы?
15. Назовите специальные переменные языка bash и их назначение.

## Лабораторная работа № 9. Программирование в командном процессоре ОС UNIX. Ветвления и циклы

### 9.1. Цель работы

Изучить основы программирования в оболочке ОС UNIX. Научится писать более сложные командные файлы с использованием логических управляющих конструкций и циклов.

### 9.2. Последовательность выполнения работы

1. Используя команды `getopts` `grep`, написать командный файл, который анализирует командную строку с ключами:
  - `-iinputfile` — прочитать данные из указанного файла;
  - `-ooutputfile` — вывести данные в указанный файл;
  - `-rшаблон` — указать шаблон для поиска;
  - `-C` — различать большие и малые буквы;
  - `-n` — выдавать номера строк.
 а затем ищет в указанном файле нужные строки, определяемые ключом `-p`.
2. Написать на языке Си программу, которая вводит число и определяет, является ли оно больше нуля, меньше нуля или равно нулю. Затем программа завершается с помощью функции `exit(n)`, передавая информацию в о коде завершения в оболочку. Командный файл должен вызывать эту программу и, проанализировав с помощью команды `$?`, выдать сообщение о том, какое число было введено.
3. Написать командный файл, создающий указанное число файлов, пронумерованных последовательно от 1 до  $N$  (например `1.tmp`, `2.tmp`, `3.tmp`, `4.tmp` и т.д.). Число файлов, которые необходимо создать, передаётся в аргументы командной строки. Этот же командный файл должен уметь удалять все созданные им файлы (если они существуют).
4. Написать командный файл, который с помощью команды `tar` запаковывает в архив все файлы в указанной директории. Модифицировать его так, чтобы запаковывались только те файлы, которые были изменены менее недели тому назад (использовать команду `find`).

### 9.3. Содержание отчёта

1. Титульный лист с указанием номера лабораторной работы и ФИО студента.
2. Формулировка цели работы.
3. Описание результатов выполнения задания:
  - скриншоты (снимки экрана), фиксирующие выполнение лабораторной работы;
  - листинги (исходный код) программ (если они есть);
  - результаты выполнения программ (текст или снимок экрана в зависимости от задания).
4. Выводы, согласованные с целью работы.
5. Ответы на контрольные вопросы.

#### 9.4. Контрольные вопросы

1. Каково предназначение команды `getopts`?
2. Какое отношение метасимволы имеют к генерации имён файлов?
3. Какие операторы управления действиями вы знаете?
4. Какие операторы используются для прерывания цикла?
5. Для чего нужны команды `false` и `true`?
6. Что означает строка `if test -f man$$/$i.$$`, встреченная в командном файле?
7. Объясните различия между конструкциями `while` и `until`.

## Лабораторная работа № 10. Программирование в командном процессоре ОС UNIX. Расширенное программирование

### 10.1. Цель работы

Изучить основы программирования в оболочке ОС UNIX. Научиться писать более сложные командные файлы с использованием логических управляющих конструкций и циклов.

### 10.2. Последовательность выполнения работы

1. Написать командный файл, реализующий упрощённый механизм семафоров. Командный файл должен в течение некоторого времени  $t_1$  дожидаться освобождения ресурса, выдавая об этом сообщение, а дождавшись его освобождения, использовать его в течение некоторого времени  $t_2 < t_1$ , также выдавая информацию о том, что ресурс используется соответствующим командным файлом (процессом). Запустить командный файл в одном виртуальном терминале в фоновом режиме, перенаправив его вывод в другой ( $> /dev/tty\#$ , где  $\#$  — номер терминала куда перенаправляется вывод), в котором также запущен этот файл, но не фоновом, а в привилегированном режиме. Доработать программу так, чтобы имела возможность взаимодействия трёх и более процессов.
2. Реализовать команду `man` с помощью командного файла. Изучите содержимое каталога `/usr/share/man/man1`. В нем находятся архивы текстовых файлов, содержащих справку по большинству установленных в системе программ и команд. Каждый архив можно открыть командой `less` сразу же просмотрев содержимое справки. Командный файл должен получать в виде аргумента командной строки название команды и в виде результата выдавать справку об этой команде или сообщение об отсутствии справки, если соответствующего файла нет в каталоге `man1`.
3. Используя встроенную переменную `$RANDOM`, напишите командный файл, генерирующий случайную последовательность букв латинского алфавита. Учтите, что `$RANDOM` выдаёт псевдослучайные числа в диапазоне от 0 до 32767.

### 10.3. Содержание отчёта

1. Титульный лист с указанием номера лабораторной работы и ФИО студента.
2. Формулировка цели работы.
3. Описание результатов выполнения задания:
  - скриншоты (снимки экрана), фиксирующие выполнение лабораторной работы;
  - листинги (исходный код) программ (если они есть);
  - результаты выполнения программ (текст или снимок экрана в зависимости от задания).
4. Выводы, согласованные с целью работы.
5. Ответы на контрольные вопросы.

### 10.4. Контрольные вопросы

1. Найдите синтаксическую ошибку в следующей строке:

```
while [$1 != "exit"]
```

2. Как объединить (конкатенация) несколько строк в одну?
3. Найдите информацию об утилите `seq`. Какими иными способами можно реализовать её функционал при программировании на `bash`?
4. Какой результат даст вычисление выражения `$((10/3))`?
5. Укажите кратко основные отличия командной оболочки `zsh` от `bash`.
6. Проверьте, верен ли синтаксис данной конструкции

```
for ((a=1; a <= LIMIT; a++))
```
7. Сравните язык `bash` с какими-либо языками программирования. Какие преимущества у `bash` по сравнению с ними? Какие недостатки?

## Лабораторная работа № 11. Средства, применяемые при разработке программного обеспечения в ОС типа UNIX/Linux

### 11.1. Цель работы

Приобрести простейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования С калькулятора с простейшими функциями.

### 11.2. Указания к лабораторной работе

#### 11.2.1. Этапы разработки приложений

Процесс разработки программного обеспечения обычно разделяется на следующие этапы:

- планирование, включающее сбор и анализ требований к функционалу и другим характеристикам разрабатываемого приложения;
- проектирование, включающее в себя разработку базовых алгоритмов и спецификаций, определение языка программирования;
- непосредственная разработка приложения:
  - кодирование — по сути создание исходного текста программы (возможно в нескольких вариантах);
  - анализ разработанного кода;
  - сборка, компиляция и разработка исполняемого модуля;
  - тестирование и отладка, сохранение произведённых изменений;
- документирование.

Для создания исходного текста программы разработчик может воспользоваться любым удобным для него редактором текста: vi, vim, mceditor, emacs, geany и др.

После завершения написания исходного кода программы (возможно состоящей из нескольких файлов), необходимо её скомпилировать и получить исполняемый модуль.

#### 11.2.2. Компиляция исходного текста и построение исполняемого файла

Стандартным средством для компиляции программ в ОС типа UNIX является GCC (GNU Compiler Collection). Это набор компиляторов для разного рода языков программирования (С, С++, Java, Фортран и др.). Работа с GCC производится при помощи одноимённой управляющей программы gcc, которая интерпретирует аргументы командной строки, определяет и осуществляет запуск нужного компилятора для входного файла.

Файлы с расширением (суффиксом) .c воспринимаются gcc как программы на языке С, файлы с расширением .cc или .C — как файлы на языке С++, а файлы с расширением .o считаются объектными.

Для компиляции файла main.c, содержащего написанную на языке С простейшую программу:

```
/*
 * main.c
 */
#include <stdio.h>
```

```
int main()
{
 printf("Hello World!\n");
 return 0;
}
```

достаточно в командной строке ввести:

```
gcc -c main.c
```

Таким образом, gcc по расширению (суффиксу) .c распознает тип файла для компиляции и формирует объектный модуль — файл с расширением .o.

Если требуется получить исполняемый файл с определённым именем (например, hello), то требуется воспользоваться опцией -o и в качестве параметра задать имя создаваемого файла:

```
gcc -o hello main.c
```

Описание некоторых опций gcc приведено в табл. 11.1.

Таблица 11.1

Некоторые опции компиляции в gcc

| Опция        | Описание                                                                                                                                                         |
|--------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -c           | компиляция без компоновки — создаются объектные файлы file.o                                                                                                     |
| -o file-name | задать имя file-name создаваемому файлу                                                                                                                          |
| -g           | поместить в файл (объектный или исполняемый) отладочную информацию для отладчика gdb                                                                             |
| -MM          | вывести зависимости от заголовочных файлов Си/или C++ программ в формате, подходящем для утилиты make; при этом объектные или исполняемые файлы не будут созданы |
| -Wall        | вывод на экран сообщений об ошибках, возникших во время компиляции                                                                                               |

Для сборки разрабатываемого приложения и собственно компиляции полезно воспользоваться утилитой make. Она позволяет автоматизировать процесс преобразования файлов программы из одной формы в другую, отслеживает взаимосвязи между файлами.

Для работы с утилитой make необходимо в корне рабочего каталога с Вашим проектом создать файл с названием makefile или Makefile, в котором будут описаны правила обработки файлов Вашего программного комплекса.

В самом простом случае Makefile имеет следующий синтаксис:

```
<цель_1> <цель_2> ... : <зависимость_1> <зависимость_2> ...
 <команда 1>
 ...
 <команда n>
```

Сначала задаётся список целей, разделённых пробелами, за которым идёт двоеточие и список зависимостей. Затем в следующих строках указываются команды. Строки с командами обязательно должны начинаться с табуляции.

В качестве цели в Makefile может выступать имя файла или название какого-то действия. Зависимость задаёт исходные параметры (условия) для достижения указанной цели. Зависимость также может быть названием какого-то действия. Команды — собственно действия, которые необходимо выполнить для достижения цели.



Рассмотрим пример Makefile для написанной выше простейшей программы, выводящей на экран приветствие 'Hello World!':

```
hello: main.c
 gcc -o hello main.c
```

Здесь в первой строке hello — цель, main.c — название файла, который мы хотим скомпилировать; во второй строке, начиная с табуляции, задана команда компиляции gcc с опциями.

Для запуска программы необходимо в командной строке набрать команду make:

```
make
Общий синтаксис Makefile имеет вид:
target1 [target2...]:[:] [dependment1...]
 [(tab) commands] [#commentary]
 [(tab) commands] [#commentary]
```

Здесь знак # определяет начало комментария (содержимое от знака # и до конца строки не будет обрабатываться. Одинарное двоеточие указывает на то, что последовательность команд должна содержаться в одной строке. Для переноса можно в длинной строке команд можно использовать обратный слэш (\). Двойное двоеточие указывает на то, что последовательность команд может содержаться в нескольких последовательных строках.

Пример более сложного синтаксиса Makefile:

```
#
Makefile for abcd.c
#

CC = gcc
CFLAGS =

Compile abcd.c normaly
abcd: abcd.c
 $(CC) -o abcd $(CFLAGS) abcd.c

clean:
 -rm abcd *.o *~

End Makefile for abcd.c
```

В этом примере в начале файла заданы три переменные: CC и CFLAGS. Затем указаны цели, их зависимости и соответствующие команды. В командах происходит обращение к значениям переменных. Цель с именем clean производит очистку каталога от файлов, полученных в результате компиляции. Для её описания использованы регулярные выражения.

### 11.2.3. Тестирование и отладка

Во время работы над кодом программы программист неизбежно сталкивается с появлением ошибок в ней. Использование отладчика для поиска и устранения ошибок в программе существенно облегчает жизнь программиста. В комплект программ GNU для ОС типа UNIX входит отладчик GDB (GNU Debugger).

Для использования GDB необходимо скомпилировать анализируемый код программы таким образом, чтобы отладочная информация содержалась в результирующем бинарном файле. Для этого следует воспользоваться опцией -g компилятора gcc:

```
gcc -c file.c -g
```

После этого для начала работы с `gdb` необходимо в командной строке ввести одноимённую команду, указав в качестве аргумента анализируемый бинарный файл:

```
gdb file.o
```

Затем можно использовать по мере необходимости различные команды `gdb`. Наиболее часто используемые команды `gdb` приведены в табл. 11.2.

Таблица 11.2

Некоторые команды `gdb`

| Команда                       | Описание действия                                                                                                                          |
|-------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| <code>backtrace</code>        | вывод на экран пути к текущей точке останова (по сути вывод названий всех функций)                                                         |
| <code>break</code>            | установить точку останова (в качестве параметра может быть указан номер строки или название функции)                                       |
| <code>clear</code>            | удалить все точки останова в функции                                                                                                       |
| <code>continue</code>         | продолжить выполнение программы                                                                                                            |
| <code>delete</code>           | удалить точку останова                                                                                                                     |
| <code>display</code>          | добавить выражение в список выражений, значения которых отображаются при достижении точки останова программы                               |
| <code>finish</code>           | выполнить программу до момента выхода из функции                                                                                           |
| <code>info breakpoints</code> | вывести на экран список используемых точек останова                                                                                        |
| <code>info watchpoints</code> | вывести на экран список используемых контрольных выражений                                                                                 |
| <code>list</code>             | вывести на экран исходный код (в качестве параметра может быть указано название файла и через двоеточие номера начальной и конечной строк) |
| <code>next</code>             | выполнить программу пошагово, но без выполнения вызываемых в программе функций                                                             |
| <code>print</code>            | вывести значение указываемого в качестве параметра выражения                                                                               |
| <code>run</code>              | запуск программы на выполнение                                                                                                             |
| <code>set</code>              | установить новое значение переменной                                                                                                       |
| <code>step</code>             | пошаговое выполнение программы                                                                                                             |
| <code>watch</code>            | установить контрольное выражение, при изменении значения которого программа будет остановлена                                              |

Для выхода из `gdb` можно воспользоваться командой `quit` (или её сокращённым вариантом `q`) или комбинацией клавиш `Ctrl-d`. Более подробную информацию по работе с `gdb` можно получить с помощью команд `gdb -h` и `man gdb`.

#### 11.2.4. Анализ исходного текста программы

Ещё одним средством проверки исходных кодов программ, написанных на языке C, является утилита `splint`. Эта утилита анализирует программный код, проверяет

корректность задания аргументов использованных в программе функций и типов возвращаемых значений, обнаруживает синтаксические и семантические ошибки.

В отличие от компилятора C анализатор splint генерирует комментарии с описанием разбора кода программы и осуществляет общий контроль, обнаруживая такие ошибки, как одинаковые объекты, определённые в разных файлах, или объекты, чьи значения не используются в работе программы, переменные с некорректно заданными значениями и типами и многое другое.

Рассмотрим следующий небольшой пример:

```
float sum(float x, float y){
 return x + y;
}

int main(){
 int x, y;
 float z;
 x = 10;
 y = 12;
 z = x + y + sum(x, y);
 return z;
}
```

Сохраните код данного примера в файл `example.c`. Для анализа кода программы следует выполнить следующую команду:

```
splint example.c
```

В результате на экран будут выведены следующие пять предупреждений

```
Splint 3.1.2 --- 03 May 2009
```

```
example.c: (in function main)
example.c:10:18: Function sum expects arg 1 to be float gets int: x
 To allow all numeric types to match, use +relaxtypes.
example.c:10:21: Function sum expects arg 2 to be float gets int: y
example.c:10:2: Assignment of int to float: z = x + y + sum(x, y)
example.c:11:9: Return value type float does not match declared
 type int: z
```

```
Finished checking --- 4 code warnings
```

Первые два предупреждения относятся к функции `sum`, которая определена для двух аргументов вещественного типа, но при вызове ей передаются два аргумента `x` и `y` целого типа. Третье предупреждение относится к присвоению вещественной переменной `z` значения целого типа, которое получается в результате суммирования `x + y + sum(x, y)`. Далее вещественное число `z` возвращается в качестве результата работы функции `main`, хотя данная функция объявлена как функция, возвращающая целое значение.

В качестве упражнения попробуйте скомпилировать программу компилятором `gcc` и сравнить выдаваемые им предупреждения с приведёнными выше.

### 11.3. Последовательность выполнения работы

1. В домашнем каталоге создайте подкаталог `~/work/os/lab_prog`.

2. Создайте в нём файлы: `calculate.h`, `calculate.c`, `main.c`.

Это будет примитивнейший калькулятор, способный складывать, вычитать, умножать и делить, возводить число в степень, брать квадратный корень, вычислять `sin`, `cos`, `tan`. При запуске он будет запрашивать первое число, операцию, второе число. После этого программа выведет результат и остановится.

Реализация функций калькулятора в файле calculate.h:

```
////////////////////////////////////
// calculate.c

#include <stdio.h>
#include <math.h>
#include <string.h>
#include "calculate.h"

float
Calculate(float Numeral, char Operation[4])
{
 float SecondNumeral;
 if(strncmp(Operation, "+", 1) == 0)
 {
 printf("Второе слагаемое: ");
 scanf("%f", &SecondNumeral);
 return(Numeral + SecondNumeral);
 }
 else if(strncmp(Operation, "-", 1) == 0)
 {
 printf("Вычитаемое: ");
 scanf("%f", &SecondNumeral);
 return(Numeral - SecondNumeral);
 }
 else if(strncmp(Operation, "*", 1) == 0)
 {
 printf("Множитель: ");
 scanf("%f", &SecondNumeral);
 return(Numeral * SecondNumeral);
 }
 else if(strncmp(Operation, "/", 1) == 0)
 {
 printf("Делитель: ");
 scanf("%f", &SecondNumeral);
 if(SecondNumeral == 0)
 {
 printf("Ошибка: деление на ноль! ");
 return(HUGE_VAL);
 }
 else
 return(Numeral / SecondNumeral);
 }
 else if(strncmp(Operation, "pow", 3) == 0)
 {
 printf("Степень: ");
 scanf("%f", &SecondNumeral);
 return(pow(Numeral, SecondNumeral));
 }
 else if(strncmp(Operation, "sqrt", 4) == 0)
 return(sqrt(Numeral));
 else if(strncmp(Operation, "sin", 3) == 0)
```

```

 return(sin(Numeral));
else if(strncmp(Operation, "cos", 3) == 0)
 return(cos(Numeral));
else if(strncmp(Operation, "tan", 3) == 0)
 return(tan(Numeral));
else
{
 printf("Неправильно введено действие ");
 return(HUGE_VAL);
}
}

```

Интерфейсный файл calculate.h, описывающий формат вызова функции-калькулятора:

```

////////////////////////////////////
// calculate.h

#ifndef CALCULATE_H
#define CALCULATE_H

float Calculate(float Numeral, char Operation[4]);

#endif /*CALCULATE_H*/

```

Основной файл main.c, реализующий интерфейс пользователя к калькулятору:

```

////////////////////////////////////
// main.c

#include <stdio.h>
#include "calculate.h"

int
main (void)
{
 float Numeral;
 char Operation[4];
 float Result;
 printf("Число: ");
 scanf("%f", &Numeral);
 printf("Операция (+, -, *, /, pow, sqrt, sin, cos, tan): ");
 scanf("%s", &Operation);
 Result = Calculate(Numeral, Operation);
 printf("%6.2f\n", Result);
 return 0;
}

```

3. Выполните компиляцию программы посредством gcc:

```

gcc -c calculate.c
gcc -c main.c
gcc calculate.o main.o -o calcul -lm

```

4. При необходимости исправьте синтаксические ошибки.
5. Создайте Makefile со следующим содержанием:

```

#
Makefile
#

CC = gcc
CFLAGS =
LIBS = -lm

calcul: calculate.o main.o
 gcc calculate.o main.o -o calcul $(LIBS)

calculate.o: calculate.c calculate.h
 gcc -c calculate.c $(CFLAGS)

main.o: main.c calculate.h
 gcc -c main.c $(CFLAGS)

clean:
 -rm calcul *.o *~

End Makefile

```

Поясните в отчёте его содержание.

6. С помощью gdb выполните отладку программы calcul (перед использованием gdb исправьте Makefile):

- Запустите отладчик GDB, загрузив в него программу для отладки:  
gdb ./calcul
- Для запуска программы внутри отладчика введите команду run:
- Для постраничного (по 9 строк) просмотра исходного код используйте команду list:
- Для просмотра строк с 12 по 15 основного файла используйте list с параметрами:  
list 12,15
- Для просмотра определённых строк не основного файла используйте list с параметрами:  
list calculate.c:20,29
- Установите точку останова в файле calculate.c на строке номер 21:  
list calculate.c:20,27  
break 21
- Выведите информацию об имеющихся в проекте точка останова:  
info breakpoints
- Запустите программу внутри отладчика и убедитесь, что программа остановится в момент прохождения точки останова:  
run  
5  
-  
backtrace
- Отладчик выдаст следующую информацию:  
#0 Calculate (Numeral=5, Operation=0x7fffffff280 "-")  
at calculate.c:21  
#1 0x00000000400b2b in main () at main.c:17

а команда `backtrace` покажет весь стек вызываемых функций от начала программы до текущего места.

- Посмотрите, чему равно на этом этапе значение переменной `Numeral`, введя:

```
print Numeral
```

На экран должно быть выведено число 5.

- Сравните с результатом вывода на экран после использования команды:

```
display Numeral
```

- Уберите точки останова:

```
info breakpoints
```

```
delete 1
```

7. С помощью утилиты `splint` попробуйте проанализировать коды файлов `calculate.c` и `main.c`.

## 11.4. Содержание отчёта

1. Титульный лист с указанием номера лабораторной работы и ФИО студента.
2. Формулировка цели работы.
3. Описание результатов выполнения задания:
  - скриншоты (снимки экрана), фиксирующие выполнение лабораторной работы;
  - листинги (исходный код) программ (если они есть);
  - результаты выполнения программ (текст или снимок экрана в зависимости от задания).
4. Выводы, согласованные с целью работы.
5. Ответы на контрольные вопросы.

## 11.5. Контрольные вопросы

1. Как получить информацию о возможностях программ `gcc`, `make`, `gdb` и др.?
2. Назовите и дайте краткую характеристику основным этапам разработки приложений в UNIX.
3. Что такое суффикс в контексте языка программирования? Приведите примеры использования.
4. Каково основное назначение компилятора языка C в UNIX?
5. Для чего предназначена утилита `make`?
6. Приведите пример структуры `Makefile`. Дайте характеристику основным элементам этого файла.
7. Назовите основное свойство, присущее всем программам отладки. Что необходимо сделать, чтобы его можно было использовать?
8. Назовите и дайте основную характеристику основным командам отладчика `gdb`.
9. Опишите по шагам схему отладки программы, которую Вы использовали при выполнении лабораторной работы.
10. Прокомментируйте реакцию компилятора на синтаксические ошибки в программе при его первом запуске.
11. Назовите основные средства, повышающие понимание исходного кода программы.
12. Каковы основные задачи, решаемые программой `splint`?

## Лабораторная работа № 12. Управление версиями

### 12.1. Цель работы

Изучить идеологию и применение средств контроля версий.

### 12.2. Системы контроля версий. Общие понятия

*Системы контроля версий (Version Control System, VCS)* применяются при работе нескольких человек над одним проектом. Обычно основное дерево проекта хранится в локальном или удалённом репозитории, к которому настроен доступ для участников проекта. При внесении изменений в содержание проекта система контроля версий позволяет их фиксировать, совмещать изменения, произведённые разными участниками проекта, производить откат к любой более ранней версии проекта, если это требуется.

В классических системах контроля версий используется централизованная модель, предполагающая наличие единого репозитория для хранения файлов. Выполнение большинства функций по управлению версиями осуществляется специальным сервером. Участник проекта (пользователь) перед началом работы посредством определённых команд получает нужную ему версию файлов. После внесения изменений, пользователь размещает новую версию в хранилище. При этом предыдущие версии не удаляются из центрального хранилища и к ним можно вернуться в любой момент. Сервер может сохранять не полную версию изменённых файлов, а производить так называемую дельта-компрессию — сохранять только изменения между последовательными версиями, что позволяет уменьшить объём хранимых данных.

Системы контроля версий поддерживают возможность отслеживания и разрешения конфликтов, которые могут возникнуть при работе нескольких человек над одним файлом. Можно объединить (слить) изменения, сделанные разными участниками (автоматически или вручную), вручную выбрать нужную версию, отменить изменения вовсе или заблокировать файлы для изменения. В зависимости от настроек блокировка не позволяет другим пользователям получить рабочую копию или препятствует изменению рабочей копии файла средствами файловой системы ОС, обеспечивая таким образом, привилегированный доступ только одному пользователю, работающему с файлом.

Системы контроля версий также могут обеспечивать дополнительные, более гибкие функциональные возможности. Например, они могут поддерживать работу с несколькими версиями одного файла, сохраняя общую историю изменений до точки ветвления версий и собственные истории изменений каждой ветви. Кроме того, обычно доступна информация о том, кто из участников, когда и какие изменения вносил. Обычно такого рода информация хранится в журнале изменений, доступ к которому можно ограничить.

В отличие от классических, в распределённых системах контроля версий центральный репозиторий не является обязательным.

Среди классических VCS наиболее известны CVS, Subversion, а среди распределённых — Git, Bazaar, Mercurial. Принципы их работы схожи, отличаются они в основном синтаксисом используемых в работе команд.



## 12.3. Указания к лабораторной работе

Система контроля версий Git представляет собой набор программ командной строки. Доступ к ним можно получить из терминала посредством ввода команды `git` с различными опциями.

Благодаря тому, что Git является распределённой системой контроля версий, резервную копию локального хранилища можно сделать простым копированием или архивацией.

### 12.3.1. Основные команды git

Наиболее часто используемые команды `git`:

- создание основного дерева репозитория:  
`git init`
- получение обновлений (изменений) текущего дерева из центрального репозитория:  
`git pull`
- отправка всех произведённых изменений локального дерева в центральный репозиторий:  
`git push`
- просмотр списка изменённых файлов в текущей директории:  
`git status`
- просмотр текущих изменений:  
`git diff`
- сохранение текущих изменений:
  - добавить все изменённые и/или созданные файлы и/или каталоги:  
`git add .`
  - добавить конкретные изменённые и/или созданные файлы и/или каталоги:  
`git add имена_файлов`
  - удалить файл и/или каталог из индекса репозитория (при этом файл и/или каталог остаётся в локальной директории):  
`git rm имена_файлов`
- сохранение добавленных изменений:
  - сохранить все добавленные изменения и все изменённые файлы:  
`git commit -am 'Описание коммита'`
  - сохранить добавленные изменения с внесением комментария через встроенный редактор:  
`git commit`
- создание новой ветки, базирующейся на текущей:  
`git checkout -b имя_ветки`
- переключение на некоторую ветку:  
`git checkout имя_ветки`  
(при переключении на ветку, которой ещё нет в локальном репозитории, она будет создана и связана с удалённой)
- отправка изменений конкретной ветки в центральный репозиторий:  
`git push origin имя_ветки`
- слияние ветки с текущим деревом:  
`git merge --no-ff имя_ветки`
- удаление ветки:
  - удаление локальной уже слитой с основным деревом ветки:  
`git branch -d имя_ветки`
  - принудительное удаление локальной ветки:

```
git branch -D имя_ветки
– удаление ветки с центрального репозитория:
git push origin :имя_ветки
```

### 12.3.2. Стандартные процедуры работы при наличии центрального репозитория

Работа пользователя со своей веткой начинается с проверки и получения изменений из центрального репозитория (при этом в локальное дерево до начала этой процедуры не должно было вноситься изменений):

```
git checkout master
git pull
git checkout -b имя_ветки
```

Затем можно вносить изменения в локальном дереве и/или ветке.

После завершения внесения какого-то изменения в файлы и/или каталоги проекта необходимо разместить их в центральном репозитории. Для этого необходимо проверить, какие файлы изменились к текущему моменту:

```
git status
```

и при необходимости удаляем лишние файлы, которые не хотим отправлять в центральный репозиторий.

Затем полезно просмотреть текст изменений на предмет соответствия правилам ведения чистых коммитов:

```
git diff
```

Если какие-либо файлы не должны попасть в коммит, то помечаем только те файлы, изменения которых нужно сохранить. Для этого используем команды добавления и/или удаления с нужными опциями:

```
git add ...
git rm ...
```

Если нужно сохранить все изменения в текущем каталоге, то используем:

```
git add .
```

Затем сохраняем изменения, поясняя, что было сделано:

```
git commit -am "Some commit message"
```

и отправляем в центральный репозиторий:

```
git push origin имя_ветки
или
git push
```

### 12.3.3. Работа с локальным репозиторием

Создадим локальный репозиторий.

Сначала сделаем предварительную конфигурацию, указав имя и email владельца репозитория:

```
git config --global user.name "Имя Фамилия"
git config --global user.email "work@mail"
```

и настроив utf-8 в выводе сообщений git:

```
git config --global core.quotePath false
```

Для инициализации локального репозитория, расположенного, например, в каталоге ~/tutorial, необходимо ввести в командной строке:

```
cd
mkdir tutorial
cd tutorial
git init
```

После это в каталоге `tutorial` появится каталог `.git`, в котором будет храниться история изменений.

Создадим тестовый текстовый файл `hello.txt` и добавим его в локальный репозиторий:

```
echo 'hello world' > hello.txt
git add hello.txt
git commit -am 'Новый файл'
```

Воспользуемся командой `status` для просмотра изменений в рабочем каталоге, сделанных с момента последней ревизии:

```
git status
```

Во время работы над проектом так или иначе могут создаваться файлы, которые не требуется добавлять в последствии в репозиторий. Например, временные файлы, создаваемые редакторами, или объектные файлы, создаваемые компиляторами. Можно прописать шаблоны игнорируемых при добавлении в репозиторий типов файлов в файл `.gitignore` с помощью сервисов. Для этого сначала нужно получить список имеющихся шаблонов:

```
curl -L -s https://www.gitignore.io/api/list
```

Затем скачать шаблон, например, для C и C++

```
curl -L -s https://www.gitignore.io/api/c >> .gitignore
```

```
curl -L -s https://www.gitignore.io/api/c++ >> .gitignore
```

### 12.3.4. Работа с сервером репозитория

Для последующей идентификации пользователя на сервере репозитория необходимо сгенерировать пару ключей (приватный и открытый):

```
ssh-keygen -C "Имя Фамилия <work@mail>"
```

Ключи сохраняются в каталоге `~/.ssh/`.

Существует несколько доступных серверов репозитория с возможностью бесплатного размещения данных. Например, <http://bitbucket.org/>.

Для работы с ним необходимо сначала зайти на сайте <http://bitbucket.org/> учётную запись. Затем необходимо загрузить сгенерённый нами ранее открытый ключ. Для этого зайти на сайт <http://bitbucket.org/> под своей учётной записью и перейти в меню [Bitbucket setting](#). После этого выбрать в боковом меню [Bitbucket setting](#) **SSH-ключи** и нажать кнопку [Добавить ключ](#). Скопировав из локальной консоли ключ в буфер обмена

```
cat ~/.ssh/id_rsa.pub | xclip -sel clip
```

вставляем ключ в появившееся на сайте поле.

После этого можно создать на сайте репозиторий, выбрав в меню [Репозитории](#)

[Создать репозиторий](#), дать ему название и сделать общедоступным (публичным).

Для загрузки репозитория из локального каталога на сервер выполняем следующие команды:

```
git remote add origin
ssh://git@bitbucket.org/<username>/<reponame>.git
git push -u origin master
```

Далее на локальном компьютере можно выполнять стандартные процедуры для работы с `git` при наличии центрального репозитория.

## 12.4. Последовательность выполнения работы

1. Настройте систему контроля версий `git`, как это описано выше с использованием сервера репозитория <http://bitbucket.org/>.
2. Для создания репозитория используйте файлы из лабораторной работы № 11: `calculate.h`, `calculate.c`, `main.c`.
3. Внесите какие-либо изменения в файлы проекта.
4. Добавьте изменения в локальный и удалённый репозитории, используя соответствующие команды `git`. Подтвердите появление изменений на сервере репозитория, сделав соответствующий скриншот.

## 12.5. Содержание отчёта

1. Титульный лист с указанием номера лабораторной работы и ФИО студента.
2. Формулировка цели работы.
3. Описание результатов выполнения задания:
  - скриншоты (снимки экрана), фиксирующие выполнение лабораторной работы;
  - листинги (исходный код) программ (если они есть);
  - результаты выполнения программ (текст или снимок экрана в зависимости от задания).
4. Выводы, согласованные с целью работы.
5. Ответы на контрольные вопросы.

## 12.6. Контрольные вопросы

1. Что такое системы контроля версий (VCS) и для решения каких задач они предназначены?
2. Объясните следующие понятия VCS и их отношения: хранилище, `commit`, история, рабочая копия.
3. Что представляют собой и чем отличаются централизованные и децентрализованные VCS? Приведите примеры VCS каждого вида.
4. Опишите действия с VCS при единоличной работе с хранилищем.
5. Опишите порядок работы с общим хранилищем VCS.
6. Каковы основные задачи, решаемые инструментальным средством `git`?
7. Назовите и дайте краткую характеристику командам `git`.
8. Приведите примеры использования при работе с локальным и удалённым репозиториями.
9. Что такое и зачем могут быть нужны ветви (branches)?
10. Как и зачем можно игнорировать некоторые файлы при `commit`?

## Лабораторная работа № 13. Именованные каналы

### 13.1. Цель работы

Приобретение практических навыков работы с именованными каналами.

### 13.2. Указания к работе

Одним из видов взаимодействия между процессами в операционных системах является обмен сообщениями. Под сообщением понимается последовательность байтов, передаваемая от одного процесса другому.

В операционных системах типа UNIX есть 3 вида межпроцессорных взаимодействий: общелинукские (именованные каналы, сигналы), System V Interface Definition (SVID — разделяемая память, очередь сообщений, семафоры) и BSD (сокет).

Для передачи данных между неродственными процессами можно использовать механизм *именованных каналов* (named pipes). Данные передаются по принципу *FIFO* (*First In First Out*) (*первым записан — первым прочитан*), поэтому они называются также *FIFO pipes* или просто *FIFO*. Именованные каналы отличаются от неименованных наличием идентификатора канала, который представлен как специальный файл (соответственно имя именованного канала — это имя файла). Поскольку файл находится на локальной файловой системе, данное IPC используется внутри одной системы.

Файлы именованных каналов создаются функцией `mkfifo(3)`.

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

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

```
int mkfifo(const char *pathname, mode_t mode);
```

Первый параметр — имя файла, идентифицирующего канал, второй параметр — маска прав доступа к файлу.

После создания файла канала процессы, участвующие в обмене данными, должны открыть этот файл либо для записи, либо для чтения. При закрытии файла сам канал продолжает существовать. Для того чтобы закрыть сам канал, нужно удалить его файл, например с помощью вызова `unlink(2)`.

Рассмотрим работу именованного канала на примере системы клиент–сервер. Сервер создаёт канал, читает из него текст, посылаемый клиентом, и выводит его на терминал.

Вызов функции `mkfifo()` создаёт файл канала (с именем, заданным макросом `FIFO_NAME`):

```
mkfifo(FIFO_NAME, 0600);
```

В качестве маски доступа используется восьмеричное значение `0600`, разрешающее процессу с аналогичными реквизитами пользователя чтение и запись. Можно также установить права доступа `0666`.

Открываем созданный файл для чтения:

```
f = fopen(FIFO_NAME, O_RDONLY);
```

Ждём сообщение от клиента. Сообщение читаем с помощью функции `read()` и печатаем на экран. После этого удаляется файл `FIFO_NAME` и сервер прекращает работу.

Клиент открывает FIFO для записи как обычный файл:

```
f = fopen(FIFO_NAME, O_WRONLY);
```

Посылаем сообщение серверу с помощью функции `write()`.

Для создания файла FIFO можно использовать более общую функцию `mknod(2)`, предназначенную для создания специальных файлов различных типов (FIFO, сокеты, файлы устройств и обычные файлы для хранения данных).

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

```
int mknod(const char *pathname, mode_t mode, dev_t dev);
```

Тогда, вместо

```
mkfifo(FIFO_NAME, 0600);
```

пишем

```
mknod(FIFO_NAME, S_IFIFO | 0600, 0);
```

Каналы представляют собой простое и удобное средство передачи данных, которое, однако, подходит не во всех ситуациях. Например, с помощью каналов довольно трудно организовать обмен асинхронными сообщениями между процессами.

### 13.3. Пример программы

#### 13.3.1. Файл `common.h`

```
/*
 * common.h - заголовочный файл со стандартными определениями
 */

#ifndef __COMMON_H__
#define __COMMON_H__

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

#define FIFO_NAME "/tmp/fifo"
#define MAX_BUFF 80

#endif /* __COMMON_H__ */
```

#### 13.3.2. Файл `server.c`

```
/*
 * server.c - реализация сервера
 *
 * чтобы запустить пример, необходимо:
 * 1. запустить программу server на одной консоли;
 * 2. запустить программу client на другой консоли.
 */
```

```
#include "common.h"

int
main()
{
 int readfd; /* дескриптор для чтения из FIFO */
 int n;
 char buff[MAX_BUFF]; /* буфер для чтения данных из FIFO */

 /* Баннер */
 printf("FIFO Server...\n");

 /* создаем файл FIFO с открытыми для всех
 * правами доступа на чтение и запись
 */
 if(mknod(FIFO_NAME, S_IFIFO | 0666, 0) < 0)
 {
 fprintf(stderr, "%s: Невозможно создать FIFO (%s)\n",
 __FILE__, strerror(errno));
 exit(-1);
 }

 /* откроем FIFO на чтение */
 if((readfd = open(FIFO_NAME, O_RDONLY)) < 0)
 {
 fprintf(stderr, "%s: Невозможно открыть FIFO (%s)\n",
 __FILE__, strerror(errno));
 exit(-2);
 }

 /* читаем данные из FIFO и выводим на экран */
 while((n = read(readfd, buff, MAX_BUFF)) > 0)
 {
 if(write(1, buff, n) != n)
 {
 fprintf(stderr, "%s: Ошибка вывода (%s)\n",
 __FILE__, strerror(errno));
 exit(-3);
 }
 }

 close(readfd); /* закроем FIFO */

 /* удалим FIFO из системы */
 if(unlink(FIFO_NAME) < 0)
 {
 fprintf(stderr, "%s: Невозможно удалить FIFO (%s)\n",
 __FILE__, strerror(errno));
 exit(-4);
 }

 exit(0);
}
```

### 13.3.3. Файл client.c

```

/*
 * client.c - реализация клиента
 *
 * чтобы запустить пример, необходимо:
 * 1. запустить программу server на одной консоли;
 * 2. запустить программу client на другой консоли.
 */

#include "common.h"

#define MESSAGE "Hello Server!!!\n"

int
main()
{
 int writefd; /* дескриптор для записи в FIFO */
 int msglen;

 /* баннер */
 printf("FIFO Client...\n");

 /* получим доступ к FIFO */
 if((writefd = open(FIFO_NAME, O_WRONLY)) < 0)
 {
 fprintf(stderr, "%s: Невозможно открыть FIFO (%s)\n",
 __FILE__, strerror(errno));
 exit(-1);
 }

 /* передадим сообщение серверу */
 msglen = strlen(MESSAGE);
 if(write(writefd, MESSAGE, msglen) != msglen)
 {
 fprintf(stderr, "%s: Ошибка записи в FIFO (%s)\n",
 __FILE__, strerror(errno));
 exit(-2);
 }

 /* закроем доступ к FIFO */
 close(writefd);

 exit(0);
}

```

### 13.3.4. Файл Makefile

```

all: server client

server: server.c common.h
 gcc server.c -o server

```



```
client: client.c common.h
 gcc client.c -o client

clean:
 -rm server client *.o
```

### 13.4. Последовательность выполнения работы

Изучите приведённые в тексте программы `server.c` и `client.c`. Взяв данные примеры за образец, напишите аналогичные программы, внося следующие изменения:

1. Работает не 1 клиент, а несколько (например, два).
2. Клиенты передают текущее время с некоторой периодичностью (например, раз в пять секунд). Используйте функцию `sleep()` для приостановки работы клиента.
3. Сервер работает не бесконечно, а прекращает работу через некоторое время (например, 30 сек). Используйте функцию `clock()` для определения времени работы сервера. Что будет в случае, если сервер завершит работу, не закрыв канал?

### 13.5. Содержание отчёта

1. Титульный лист с указанием номера лабораторной работы и ФИО студента.
2. Формулировка цели работы.
3. Описание результатов выполнения задания:
  - скриншоты (снимки экрана), фиксирующие выполнение лабораторной работы;
  - листинги (исходный код) программ (если они есть);
  - результаты выполнения программ (текст или снимок экрана в зависимости от задания).
4. Выводы, согласованные с целью работы.
5. Ответы на контрольные вопросы.

### 13.6. Контрольные вопросы

1. В чем ключевое отличие именованных каналов от неименованных?
2. Возможно ли создание неименованного канала из командной строки?
3. Возможно ли создание именованного канала из командной строки?
4. Опишите функцию языка С, создающую неименованный канал.
5. Опишите функцию языка С, создающую именованный канал.
6. Что будет в случае прочтения из `fifo` меньшего числа байтов, чем находится в канале? Большого числа байтов?
7. Аналогично, что будет в случае записи в `fifo` меньшего числа байтов, чем позволяет буфер? Большого числа байтов?
8. Могут ли два и более процессов читать или записывать в канал?
9. Опишите функцию `write` (тип возвращаемого значения, аргументы и логику работы). Что означает 1 (единица) в вызове этой функции в программе `server.c` (строка 42)?
10. Опишите функцию `strerror`.

## Лабораторная работа № 14. Очереди сообщений

### 14.1. Цель работы

Приобретение практических навыков работы с очередями сообщений.

### 14.2. Указания к работе

Межпроцессорное взаимодействие типа «очередь сообщений» обладает следующими свойствами:

- накопление сообщений в очереди;
- извлечение сообщений из очереди в произвольно заданном порядке;
- обработка сообщений произвольной структуры и заданного размера.

Приложения, работающие с очередями сообщений, могут формировать очередь и обрабатывать в ней сообщения, используя идентификаторы для обращения к конкретным сообщениям. Это позволяет использовать приоритетную обработку сообщений, а также различать сообщения, формируемые разными приложениями, принимающими участие в обмене сообщениями.

Обработка сообщений, поступающих в очередь, может осуществляться последовательно или в произвольном порядке. В тоже время группа процессов, выступающая в качестве генератора сообщений, может создавать как одну, так и несколько очередей для обмена сообщениями. Формируемая очередь может также использоваться как одним, так и несколькими процессами. Максимальный размер сообщения и максимальное количество сообщений в очереди ограничены<sup>1</sup>.

Для определения структуры данных можно использовать следующую конструкцию:

```
struct my_msgbuf
{
 long mtype;
 ...
};
```

Поле `mtype` является обязательным и содержит идентификатор очереди сообщения. Кроме этого поля в структуре данных сообщения могут быть описаны другие поля произвольного типа.

В System V Interface Definition взаимодействие клиента и сервера идентифицируется по ключу. Ключ генерируется на основе имени файла (берётся его `i-node`). Для генерации нескольких ключей на основе одного имени файла используется дополнительный *идентификатор проекта* (обычно ASCII-символ).

Обычно ключ можно задать самостоятельно либо использовать функцию `ftok(3)`:

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

key_t ftok(const char *pathname, int proj_id);
```

---

<sup>1</sup> В ОС Linux в файле `<linux/msg.h>` максимальная длина сообщения задаётся константой `MSGMAX` и измеряется в байтах, а максимальное число сообщений — определяется константой `MSGMNG`. В ОС на базе платформы IA32 размер сообщения ограничен 8 килобайтами, а длина очереди — 16384 (16K) сообщениями.

Но нужно понимать, что никакой гарантии уникальности этого ключа не существует.

Создать очередь сообщений (или подключиться к ней) можно с помощью функции `msgget()`:

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

```
int msgget(key_t key, int msgflg)
```

Первый параметр `msgget()` — ключ (предназначен только для открытия очереди), второй параметр — комбинация маски прав доступа и дополнительных флагов (например, флаг `IPC_CREATE` указывает, что нужно создать новую очередь, а флаг `IPC_EXCL` проверяет, существует ли очередь с указанным ключом).

С помощью функций `msgsnd()` и `msgrcv()` осуществляется передача и получение сообщений:

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

```
int msgsnd(int msqid, const void *msgp, size_t msgsz,
 int msgflg);
```

```
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz,
 long msgtyp,
 int msgflg);
```

Параметрами этих функций являются идентификатор очереди (который можно получить из функции `msgget()`), указатель на структуру сообщения, размер структуры сообщения, дополнительные флаги.

В функции `msgrcv()` четвёртым параметром идёт идентификатор сообщения, в зависимости от значения которого определяется принцип извлечения сообщений из очереди:

- из очереди будет извлечено сообщение со значением поля `mtype`, если значение этого параметра больше нуля;
- из очереди будет извлечено первое по порядку сообщение, если этот параметр равен нулю;
- из очереди будет извлечено первое сообщение, чей идентификатор меньше либо равен абсолютному значению параметра, если этот параметр отрицательный.

С помощью функции `msgctl()` задаётся управление очередью сообщений:

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

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Параметрами этой функции являются идентификатор очереди, команда (`IPC_STAT`, `IPC_SET` или `IPC_RMID`), указатель на структуру `msqid_ds` (поля структуры содержат значения параметров очереди). Последний параметр используется в вызовах-запросах (в `msgctl()` используется команда `IPC_STAT`) и для конфигурации очереди (в `msgctl()` используется команда `IPC_SET`).

Для удаления очереди с идентификатором `msqid` следует использовать:

```
msgctl(msqid, IPC_RMID, 0);
```

### 14.3. Пример программы

#### 14.3.1. Файл common.h

```
/*
 * common.h - заголовочный файл со стандартными определениями
 */

#ifndef __COMMON_H__
#define __COMMON_H__

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAXBUF 80
#define PERM 0666

/*
 * Определение структуры сообщения. Первым элементом структуры
 * должен быть элемент типа long, указывающий на тип сообщения.
 * Другие элементы могут быть определены дополнительно.
 */

typedef struct my_msgbuf
{
 long mtype;
 char buff[MAXBUF];
} my_message_t;

#endif /* __COMMON_H__ */
```

#### 14.3.2. Файл server.c

```
/*
 * server.c - реализация сервера
 *
 * чтобы запустить пример необходимо:
 * 1. запустить программу server на одной консоли;
 * 2. запустить программу client на другой консоли.
 */

#include "common.h"

int
main()
{
 my_message_t message;
 key_t key;
```

```
int msgid;
int length;
int n;

/* баннер */
printf("Message Queue Server...\n");

/* получить ключ */
if((key = ftok("server", 'A')) < 0)
{
 fprintf(stderr,
 "%s: Невозможно получить ключ (%s)\n",
 __FILE__, strerror(errno));
 exit(-1);
}

/* создать очередь сообщений */
if((msgid = msgget(key, PERM | IPC_CREAT)) < 0)
{
 fprintf(stderr,
 "%s: Невозможно создать очередь (%s)\n",
 __FILE__, strerror(errno));
 exit(-1);
}

/* прочитать сообщение */
message.mtype = 1L;
n = msgrcv(msgid, &message, sizeof(message), message.mtype, 0);

/* если сообщение поступило, напечатать содержимое */
if(n <= 0)
{
 fprintf(stderr,
 "%s: Ошибка чтения сообщения (%s)\n",
 __FILE__, strerror(errno));
 exit(-1);
}
else
{
 if(write(1, message.buff, n) != n)
 {
 fprintf(stderr,
 "%s: Ошибка вывода (%s)\n",
 __FILE__, strerror(errno));
 exit(-2);
 }
}

/* удалить очередь сообщений */
if(msgctl(msgid, IPC_RMID, 0) < 0)
{
 fprintf(stderr,
 "%s: Ошибка удаления очереди (%s)\n",
```

```
 __FILE__, strerror(errno));
 exit(-3);
}

exit(0);
}
```

### 14.3.3. Файл client.c

```
/*
 * client.c - реализация сервера
 *
 * чтобы запустить пример необходимо:
 * 1. запустить программу server на одной консоли;
 * 2. запустить программу client на другой консоли.
 */

#include "common.h"

#define MESSAGE "Hello world!!!\n"

int
main()
{
 my_message_t message;
 key_t key;
 int msgid;
 int length;
 int n;

 /* баннер */
 printf("Message Queue Client...\n");

 /* получить ключ */
 if((key = ftok("server", 'A')) < 0)
 {
 fprintf(stderr,
 "%s: Невозможно получить ключ (%s)\n",
 __FILE__, strerror(errno));
 exit(-1);
 }

 /* получить доступ к очереди сообщений, очередь уже должна
 * быть создана сервером
 */
 if((msgid = msgget(key, 0)) < 0)
 {
 fprintf(stderr,
 "%s: Невозможно получить доступ к очереди (%s)\n",
 __FILE__, strerror(errno));
 exit(-2);
 }
}
```

```
/* подготовка сообщения */
message.mtype = 1L;
if((length = sprintf(message.buff, MESSAGE)) < 0)
{
 fprintf(stderr,
 "%s: Ошибка копирования в буфер (%s)\n",
 __FILE__, strerror(errno));
 exit(-3);
}

/* передача сообщения */
if(msgsnd(msgid, (void*)&message, length, 0) < 0)
{
 fprintf(stderr,
 "%s: Ошибка записи сообщения в очередь (%s)\n",
 __FILE__, strerror(errno));
 exit(-4);
}

exit(0);
}
```

#### 14.3.4. Файл Makefile

```
all: server client

server: server.c common.h
 gcc server.c -o server

client: client.c common.h
 gcc client.c -o client

clean:
 -rm server client *.o
```

### 14.4. Последовательность выполнения работы

Изучите приведённые в тексте программы `server.c` и `client.c`. Взяв данные примеры за образец, напишите аналогичные программы, внося следующие изменения:

1. Работает 1 сервер и несколько клиентов;
2. Мультиплексировать сообщения в очередь. Каждый клиент посылает свой тип сообщений (`mtype` равен PID процесса) и сервер распечатывает их сообщения, указывая для каждого сообщения от какого клиента оно пришло.

### 14.5. Содержание отчёта

1. Титульный лист с указанием номера лабораторной работы и ФИО студента.
2. Формулировка цели работы.
3. Описание результатов выполнения задания:

- скриншоты (снимки экрана), фиксирующие выполнение лабораторной работы;
  - листинги (исходный код) программ (если они есть);
  - результаты выполнения программ (текст или снимок экрана в зависимости от задания).
4. Выводы, согласованные с целью работы.
  5. Ответы на контрольные вопросы.

#### 14.6. Контрольные вопросы

1. Какая система используется для именования очередей сообщений? Опишите системный вызов `ftok()`.
2. Можно ли для генерации ключа использовать файл `.bashrc`? Обоснуйте ответ.
3. Опишите разницу работы процессов с дескрипторами файлов и дескрипторами IPC. (Подсказка: могут ли разные процессы использовать одинаковые дескрипторы для доступа к файлу? А для доступа к IPC?)
4. Опишите системные вызовы создания очереди, записи сообщения в очередь, извлечения сообщения из очереди и управления очередью. Какой системный вызов позволяет удалить очередь? Опишите структуру данных сообщения.



## Лабораторная работа № 15. Сокеты

### 15.1. Цель работы

Приобретение практических навыков работы с сокетами.

### 15.2. Указания к работе

Большинство операционных систем имеет возможность реализовывать обмен данными между процессами посредством сокетов. Под сокетом понимается некий абстрактный объект, реализующий точку соединения 2-х объектов (процессов).

Принцип взаимодействия процессов через сокеты основан на файловых функциях Unix-подобных систем `read()` и `write()`. При этом передача данных может осуществляться как в синхронном, так и в асинхронном режиме.

#### 15.2.1. Работа с сокетами

Создание сокета реализуется посредством функции `socket()`:

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

```
int socket(int domain, int type, int protocol);
```

В качестве параметров этой функции передаются домен, тип сокета и протокол передачи данных.

Под доменом в функции `socket()` подразумевается тип соединения:

- `PF_UNIX`, `PF_LOCAL` — локальное соединение;
- `PF_INET` — сетевое соединение по протоколу IPv4;
- `PF_INET6` — сетевое соединение по протоколу IPv6;
- `PF_IPX` — сетевое соединение по протоколу IPX фирмы Novell;
- `PF_NETLINK` — интерфейс взаимодействия с ядром ОС;
- `PF_PACKET` — низкоуровневый пакетный интерфейс.

Тип сокета выбирается по требующемуся назначению:

- `SOCK_STREAM` — обеспечивает надёжную двунаправленную передачу потока данных с предварительным установлением соединения;
- `SOCK_DGRAM` — обеспечивает ненадёжную передачу дейтаграммных данных ограниченного объёма без предварительного установления соединения;
- `SOCK_SEQPACKET` — обеспечивает надёжную последовательную двунаправленную передачу дейтаграммных данных ограниченного объёма с предварительным установлением соединения;
- `SOCK_RAW` — обеспечивает доступ к низкоуровневому сетевому протоколу;
- `SOCK_RDM` — обеспечивает надёжную дейтаграммную передачу данных без гарантий их последовательности, но с предварительным установлением соединения.

Значение, ассоциированное с протоколом передачи данных в функции `socket()`, зависит от указанного в качестве первого параметра домена.

В качестве результата работы функции `socket()` возвращается идентификатор сокета.

В работе с дейтаграммными сокетами вместо функций `write()` и `read()` используются функции `recvfrom()` и `sendto()`.

Функция `recvfrom()` используется для чтения данных:

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

```
int recvfrom(int s, void *buf, size_t len,
 int flags, struct sockaddr *from,
 socklen_t *fromlen);
```

В качестве параметров функции `recvfrom()` указываются идентификатор сокета, указатель на буфер для получения сообщений, длина буфера, дополнительные флаги, указатель на структуру адреса клиента, и длина адреса клиента. При помощи указателя на структуру типа `sockaddr` можно получить адрес объекта, запросившего установление соединения при сетевом обмене. Ещё один параметр функции `recvfrom()` — указатель на переменную, возвращающую значение длины структуры, задающей адрес. Оба эти параметра могут иметь значение `NULL`, если при обмене посредством сокетов не требуется устанавливать сетевое соединение.

Функция `sendto()` используется для записи данных:

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

```
int sendto(int s, const void *msg, size_t len,
 int flags, const struct sockaddr *to,
 socklen_t tolen);
```

В качестве параметров функции `sendto()` указываются идентификатор сокета, адрес буфера для передачи сообщений, длина буфера, дополнительные флаги, указатель на структуру адреса объекта, с которым устанавливается соединение, и длина адреса объекта.

Для работы с потоковыми сокетами используются функция `recv(2)`:

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

```
int recv(int s, void *buf, size_t len, int flags);
и функция send(2):
```

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

```
int send(int s, const void *msg,
 size_t len, int flags);
```

Структура параметров аналогична описанным выше параметрам функций `recvfrom()` и `sendto()`.

Для закрытия сокетов (освобождения занимаемых сокетом ресурсов) используется функция `close()`.

### 15.2.2. Порядок байтов

Для того чтобы осуществить межпроцессное взаимодействие между компьютерами с разной архитектурой, необходимо нормализовать порядок байтов в передаваемых потоках.

Для преобразования порядка байтов из архитектурнозависимой формы к сетевой нормализованной форме используют специальные функции:

```
#include <arpa/inet.h>
```

```
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
```

```
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

### 15.2.3. Сокеты в файловом пространстве имён

Сокеты в файловом пространстве имён (file namespace, сокеты Unix) используются в качестве адресов имени файлов специального типа. Важной особенностью этих сокетов является то, что соединение с их помощью локального и удалённого приложений невозможно, даже если файловая система, в которой создан сокет, доступна удалённой операционной системе.

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

```
unix_socket = socket(AF_UNIX, type, 0);
```

Формат адреса имеет вид (unix(8)):

```
#define UNIX_PATH_MAX 108
```

```
struct sockaddr_un {
 sa_family_t sun_family; /* AF_UNIX */
 char sun_path[UNIX_PATH_MAX]; /* pathname */
};
```

В поле `sun_family` находится имя домена (в данном случае — `AF_UNIX`). Второе поле — имя файла, идентифицирующего сокет.

После получения дескриптора сокета вызывается функция `bind(2)`, которая связывает сокет с заданным адресом.

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

```
int bind(int sockfd, struct sockaddr *my_addr,
 socklen_t addrlen);
```

Первым параметром функции является дескриптор, вторым — указатель на структуру `sockaddr`, содержащую адрес, на котором регистрируется сервер, третий параметр функции — длина структуры, содержащей адрес.

### 15.2.4. Парные сокеты

Сокеты в файловом пространстве имён похожи на именованные каналы тем, что для идентификации сокетов используются файлы специального типа. В мире сокетов есть и аналог неименованных каналов — *парные сокеты* (*socket pairs*). Как и неименованные каналы, парные сокеты создаются парами и не имеют имён. Естественно, что область применения парных сокетов та же, что и у неименованных каналов, — взаимодействие между родительским и дочерним процессом. Так же как и в случае неименованного канала, первый из дескрипторов используется одним процессом, второй — другим.

Парные сокеты создаются функцией `socketpair(2)`

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

```
int socketpair(int domain, int type, int protocol,
 int sv[2]);
```

Первые три параметра функции те же, что и у `socket()`, а четвёртым параметром является массив из двух переменных, в которых возвращаются дескрипторы. Дескрипторы сокетов, возвращённые `socketpair()`, уже готовы к передаче данных, так что можно применять к ним функции `read()` и `write()`.

Указание домена в функции `socketpair()` выглядит явно излишне, поскольку для этой функции система поддерживает только сокеты в домене `AF_UNIX` (вполне логичное ограничение, если учесть, что парные сокеты не имеют имён и предназначены для обмена данными между родственными процессами).

### 15.2.5. Сетевые сокеты

Формат адреса имеет вид `ip(7)`:

```
struct sockaddr_in
{
 sa_family_t sin_family; /* AF_INET */
 u_int16_t sin_port; /* port */
 struct in_addr sin_addr; /* ip address */
};

struct in_addr
{
 u_int32_t s_addr;
```

Поскольку адрес транспортного уровня состоит из пары `ip-адрес:порт`, то и в структуре под адрес отводится два поля.

Сетевой сервер должен уметь выполнять запросы множества клиентов одновременно. При этом в соединениях «точка-точка», например, при использовании потоковых сокетов, у сервера для каждого клиента должен быть открыт отдельный сокет. Для этого создаётся два сокета: один для прослушивания входящих запросов, другой для их обработки (иначе все другие попытки соединиться с сервером по указанному адресу и порту будут заблокированы).

Функция `listen(2)` переводит сервер в режим ожидания запроса на соединение:

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

```
int listen(int s, int backlog);
```

Второй параметр `listen(2)` — максимальное число соединений, которые сервер может обрабатывать одновременно.

Далее вызывается функция `accept(2)`, которая устанавливает соединение в ответ на запрос клиента:

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

```
int accept(int s, struct sockaddr *addr,
 socklen_t *addrlen);
```

Второй параметр функции `accept(2)` содержит сведения об адресе клиента, запрошившего соединение, а третий параметр указывает размер этого адреса.

Получив запрос на соединение, функция `accept(2)` возвращает новый сокет, открытый для обмена данными с клиентом, запросившим соединение. Сервер как бы перенаправляет запрошенное соединение на другой сокет, оставляя первоначальный сокет свободным для прослушивания запросов на установку соединения.

Также нужно решить задачу преобразования доменного имени в сетевой адрес. Разрешение доменных имён выполняет функция `gethostbyname(3)`:

```
#include <netdb.h>
```

```
struct hostent *gethostbyname(const char *name);
```

Функция получает указатель на строку с Интернет-именем сервера и возвращает указатель на структуру `hostent`, которая содержит имя сервера в приемлемом для дальнейшего использования виде. При этом, если необходимо, выполняется разрешение доменного имени в сетевой адрес.

## 15.3. Пример программы

### 15.3.1. Файл `common.h`

```
/*
 * common.h - заголовочный файл
 * со стандартными определениями
 */

#ifndef __COMMON_H__
#define __COMMON_H__

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

#define SOCKET_NAME "/tmp/echo.server"
#define MAX_BUF 80

#endif /* __COMMON_H__ */
```

### 15.3.2. Файл `server.c`

```
/*
 * server.c - реализация сервера
 *
 * чтобы запустить пример необходимо:
 * 1. запустить программу server на одной консоли;
 * 2. запустить программу client на другой консоли.
 */

#include "common.h"

#define MESSAGE "Hello Client!!!"

int
main()
```

```
{
 char buf[MAX_BUFF];
 struct sockaddr_un serv_addr;
 struct sockaddr_un clnt_addr;
 int sockfd;
 int saddrlen;
 int max_caddrlen;
 int msglen;

 /* Баннер */
 printf("Unix Sockets Server...\n");

 /* создаем сокет */
 if((sockfd = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0)
 {
 fprintf(stderr, "%s: Невозможно создать сокет (%s)\n",
 __FILE__, strerror(errno));
 exit(-1);
 }

 /* свяжем сокет с известным локальным адресом. поскольку адрес
 * в домене UNIX представляет собой имя, котрый будет создан
 * системным вызовом bind, сначала удалим файл с этим именем в
 * случае, если он сохранился от предыдущего запуска.
 */

 unlink(SOCKET_NAME);
 bzero(&serv_addr, sizeof(serv_addr));
 serv_addr.sun_family = AF_UNIX;
 strcpy(serv_addr.sun_path, SOCKET_NAME);
 saddrlen = sizeof(serv_addr.sun_family)
 + strlen(serv_addr.sun_path);

 if(bind(sockfd,
 (struct sockaddr *)&serv_addr, saddrlen) < 0)
 {
 fprintf(stderr,
 "%s: Ошибка связывания сокета с адресом (%s)\n",
 __FILE__, strerror(errno));
 exit(-2);
 }

 /* теперь запустим бесконечный цикл
 * чтения сообщений от клиентов
 * и отправления их обратно
 */
 max_caddrlen = sizeof(clnt_addr);
 for(; ;)
 {
 int n;
 int caddrlen;

 caddrlen = max_caddrlen;
```

```
n = recvfrom(sockfd, buf, MAX_BUFF, 0,
 (struct sockaddr*)&clnt_addr,
 &caddrlen);

if (n < 0)
{
 fprintf(stderr,
 "s: Ошибка приема (%s)\n",
 __FILE__, strerror(errno));
 exit(-3);
}

/* выведем его на экран */
printf("echo: %s\n", buf);

/* отправляем данные */
msglen = strlen(MESSAGE);

/* благодаря вызову recvfrom,
 * мы знаем адрес клиента, от которого
 * получено сообщение. используем этот
 * адрес для передачи сообщения
 * обратно отправителю.
 */
if (sendto(sockfd, MESSAGE, msglen, 0,
 (struct sockaddr*)&clnt_addr,
 caddrlen) != n)
{
 fprintf(stderr,
 "s: Ошибка передачи (%s)\n",
 __FILE__, strerror(errno));
 exit(-5);
}

}

/* cleanup */
close(sockfd);
unlink(clnt_addr.sun_path);

exit(0);
}
```

### 15.3.3. Файл client.c

```
/*
 * client.c - реализация клиента
 *
 * чтобы запустить пример необходимо:
 * 1. запустить программу server на одной консоли;
 * 2. запустить программу client на другой консоли.
 */
```

```
#include "common.h"

#define MESSAGE "Hello Server!!!"

int
main()
{
 char buf[MAX_BUFF];
 struct sockaddr_un serv_addr;
 struct sockaddr_un clnt_addr;
 int sockfd;
 int saddrlen;
 int caddrlen;
 int msglen;
 int n;

 /* Установим адрес сервера, с которым
 * мы будем обмениваться данными.
 * Для этого заполним структуру данных
 * sockaddr_un, которую будем
 * использовать при отправлении
 * данных серверу с помощью вызова sendto.
 * значение адреса известно
 * по предварительной договоренности.
 */

 bzero(&serv_addr, sizeof(serv_addr));
 serv_addr.sun_family = AF_UNIX;
 strcpy(serv_addr.sun_path, SOCKET_NAME);
 saddrlen = sizeof(serv_addr.sun_family)
 + strlen(serv_addr.sun_path);

 /* Необходимо связать сокет с некоторым
 * локальным адресом, чтобы сервер
 * имел возможность возвратить
 * посланное сообщение. Этот адрес должен
 * быть уникальным в пределах
 * коммуникационного домена --- т.е. данной
 * операционной системы. Для обеспечения
 * этого условия, воспользуемся
 * функцией mkstemp, которая возвращает
 * уникальное имя, основанное на
 * представленном шаблоне и
 * идентификаторе нашего процесса PID.
 */

 bzero(&clnt_addr, sizeof(clnt_addr));
 clnt_addr.sun_family = AF_UNIX;
 strcpy(clnt_addr.sun_path, "/tmp/clnt.XXXXXX");
 mktemp(clnt_addr.sun_path);
 caddrlen = sizeof(clnt_addr.sun_family)
 + strlen(clnt_addr.sun_path);
```



```
/* создадим сокет датаграмм */
if((sockfd = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0)
{
 fprintf(stderr,
 "%s: Невозможно создать сокет (%s)\n",
 __FILE__, strerror(errno));
 exit(-1);
}

if(bind(sockfd,
 (struct sockaddr *)&clnt_addr, caddrlen) < 0)
{
 fprintf(stderr,
 "%s: Ошибка связывания сокета с адресом (%s)\n",
 __FILE__, strerror(errno));
 exit(-2);
}

/* отправляем данные */
msglen = strlen(MESSAGE);
if(sendto(sockfd, MESSAGE, msglen, 0,
 (struct sockaddr *)&serv_addr,
 saddrlen) != msglen)
{
 fprintf(stderr,
 "%s: Ошибка передачи (%s)\n",
 __FILE__, strerror(errno));
 exit(-3);
}

/* прочитаем ответ */
memset(buf, 0, MAX_BUFF);
if((n = recvfrom(sockfd,
 buf, MAX_BUFF, 0, NULL, 0)) < 0)
{
 fprintf(stderr,
 "%s: Ошибка приема (%s)\n",
 __FILE__, strerror(errno));
 exit(-4);
}

/* выведем его на экран */
printf("echo: %s\n", buf);

/* cleanup */
close(sockfd);
unlink(clnt_addr.sun_path);

exit(0);
}
```

### 15.3.4. Файл Makefile

```
CC = gcc
CFLAGS =

programs = server client

all: $(programs)

server: server.c common.h
 $(CC) $(CFLAGS) $< -o $@

client: client.c common.h
 $(CC) $(CFLAGS) $< -o $@

clean:
 -rm $(programs) *.o
```

## 15.4. Последовательность выполнения работы

Изучите приведённые в тексте программы `server.c` и `client.c`. Взяв данные примеры за образец, напишите аналогичные программы, внося следующие изменения:

1. Разобраться в предложенных программах–примерах;
2. Запускать несколько клиентов. Все ли в порядке с мультиплексированием сообщений?

## 15.5. Содержание отчёта

1. Титульный лист с указанием номера лабораторной работы и ФИО студента.
2. Формулировка цели работы.
3. Описание результатов выполнения задания:
  - скриншоты (снимки экрана), фиксирующие выполнение лабораторной работы;
  - листинги (исходный код) программ (если они есть);
  - результаты выполнения программ (текст или снимок экрана в зависимости от задания).
4. Выводы, согласованные с целью работы.
5. Ответы на контрольные вопросы.

## 15.6. Контрольные вопросы

1. Что означает аббревиатура BSD в названии?
2. Что такое сокет? В виде какого объекта он представляется в операционной системе?
3. Опишите основные отличия сокетов от остальных IPC.
4. Что такое коммуникационный домен? Опишите своими словами.
5. Какие виды сокетов бывают?
6. Опишите системный вызов `BSD socket()`.
7. Опишите схему взаимодействия между процессами при установлении виртуального канала. Опишите основные функции, которые при этом используются.

## **Учебно-методический комплекс**

Рекомендуется для направлений подготовки

01.03.02 «Прикладная математика и информатика»  
02.03.01 — «Математика и компьютерные науки»  
02.03.02 — Фундаментальная информатика и информационные технологии  
09.03.03 «Прикладная информатика»  
38.03.05 «Бизнес-информатика»

Квалификация (степень) выпускника: бакалавр



## Программа дисциплины

### Цели и задачи дисциплины

Целью дисциплины является введение учащихся в предметную область современных операционных систем.

В процессе преподавания дисциплины решаются следующие задачи:

- анализ принципов построения и архитектур операционных систем;
- обучение работе в операционной системе типа Unix.

### Место дисциплины в структуре основной образовательной программы

Дисциплина относится к базовой части блока 1 «Дисциплины (модули)» и обеспечивает изучение дисциплин, связанных с работой на компьютере, «Компьютерные сети», «Сетевые технологии», «Администрирование сетевых подсистем», «Администрирование локальных систем», «Информационная безопасность».

Требования к входным знаниям, умениям и компетенциям студента: необходима подготовка по дисциплине «Архитектура вычислительных систем».

### Требования к результатам освоения дисциплины

Процесс изучения дисциплины направлен на формирование следующих компетенций:

- для направления «Фундаментальная информатика и информационные технологии»:
  - ОПК-2: способность применять в профессиональной деятельности современные языки программирования и языки баз данных, методологии системной инженерии, системы автоматизации проектирования, электронные библиотеки и коллекции, сетевые технологии, библиотеки и пакеты программ, современные профессиональные стандарты информационных технологий;
- для направления «Математика и компьютерные науки»:
  - ОПК-2: способность решать стандартные задачи профессиональной деятельности на основе информационной и библиографической культуры с применением информационно-коммуникационных технологий и с учётом основных требований информационной безопасности;
- для направления «Прикладная математика и информатика»:
  - ОПК-1: способность использовать базовые знания естественных наук, математики и информатики, основные факты, концепции, принципы теорий, связанных с прикладной математикой и информатикой;
- для направления «Прикладная информатика»:
  - ПК-10: способность принимать участие во внедрении, адаптации и настройке информационных систем;
  - ПК-11: способность эксплуатировать и сопровождать информационные системы и сервисы;
  - ПК-13: способность осуществлять установку и настройку параметров программного обеспечения информационных систем;
- для направления «Бизнес-информатика»:

- ОПК-3: способность работать с компьютером как средством управления информацией, работать с информацией из различных источников, в том числе в глобальных компьютерных сетях;
- ПК-30: владеть базовыми знаниями и навыками использования в профессиональной деятельности операционных систем, платформенных окружений, сетевых технологий.

В результате изучения дисциплины студент должен:

**Знать:**

- общие принципы построения операционных систем;
- составные элементы ОС Unix;
- принципы функционирования системного программного обеспечения.

**Уметь:**

- работать в ОС Linux;
- программировать, используя системный API ОС Unix.

**Владеть:**

- базовыми методами администрирования операционных систем;
- базовыми навыками работы в ОС Unix.

## Объем дисциплины и виды учебной работы

Общая трудоемкость дисциплины составляет 3 зачётных единиц.

| №     | Вид учебной работы                           | Всего часов | Семестры |
|-------|----------------------------------------------|-------------|----------|
|       |                                              |             | 2        |
| 1.    | Аудиторные занятия (всего)                   | 54          | 54       |
| 1.1   | В том числе: Лекции                          | 18          | 18       |
| 1.2   | Прочие занятия:                              |             |          |
| 1.2.1 | Практические занятия (ПЗ)                    |             |          |
| 1.2.2 | Семинары (С)                                 |             |          |
| 1.2.3 | Лабораторные работы (ЛР)                     | 36          | 36       |
| 2.    | Самостоятельная работа студентов (ак. часов) | 54          | 54       |
| 3.    | Общая трудоемкость (ак. часов)               | 108         | 108      |
| 4.    | Общая трудоёмкость (зачетных единиц)         | 3           | 3        |

## Содержание дисциплины

### Содержание разделов дисциплины

#### 1. Общие принципы ОС UNIX:

- 1.1. Введение в операционную систему UNIX. Типы ОС. ОС реального времени и разделения времени. Алгоритм работы ОС реального времени и их преимущества и недостатки. Алгоритм работы ОС разделения времени и их преимущества и недостатки. Различия в ОС реального времени и разделения времени. Введение в архитектуру ОС. Архитектура монолитной ОС,

- примеры таких систем. Архитектура многоуровневой ОС, примеры. Принципы организации ОС типа виртуальной машины, примеры таких машин. Архитектура ОС типа клиент-сервер.
- 1.2. Архитектура UNIX. Файлы и устройства. Понятие виртуальной файловой системы. Функции виртуальной файловой системы Unix (VFS). Архитектура виртуальной файловой системы. Зависимости. Потоки данных. Управляющие потоки. Внешний и внутренний интерфейсы виртуальной файловой системы. Понятие драйверов файловой системы и их типы. Понятие кэша. Механизмы обмена данными в ОС. Понятие логической файловой системы. Монтирование и демонтирование. Физическая организация файловой системы. Понятие i-узлов. Типы файлов. Структура файла обычного типа. Особенности организации файловой системы Unix. Внутренняя структура виртуальной файловой системы. Зависимости виртуальной файловой системы от других подсистем ядра.
  - 1.3. Архитектура UNIX. Процессы. Понятие процесса, определение процесса, примеры процессов. Понятие примитива, определение примитива, примеры примитивов. Отличия процессов и примитивов. Понятие среды выполнения. Уровень выполнения ядра и уровень выполнения задачи. Создание процессов, управление процессами из программы пользователя.
  - 1.4. Терминал и командная строка. Эффективное использование командной строки. Справочная подсистема.
  2. Начала администрирования ОС UNIX:
    - 2.1. Введение в безопасность UNIX. Основы информационной безопасности. Концепции безопасности UNIX. Управление правами доступа.
    - 2.2. Сеть в UNIX. Сетевая подсистема. Общие принципы работы. Понятие сокетов. Типы сокетов. Общие принципы взаимодействия ОС через сокет. Интерфейс сетевой подсистемы. Архитектура сетевой подсистемы. Зависимости. Потоки данных. Управляющие потоки. Состав и описание модулей сетевой подсистемы. Зависимости сетевой подсистемы от подсистем ядра.
    - 2.3. Управление службами. Загрузка операционной системы. Системные службы. Мониторинг и журналирование.
    - 2.4. Управление программным обеспечением. Управление программным обеспечением: роли и задачи. Формы распространения программного обеспечения. Управление пакетами.

## Разделы дисциплин и виды занятий

| № п/п | Наименование раздела дисциплины  | Лекц. | Практические занятия и лабораторные работы |    |  | СРС | Всего час. |
|-------|----------------------------------|-------|--------------------------------------------|----|--|-----|------------|
|       |                                  |       | ПЗ/С                                       | ЛР |  |     |            |
| 1.    | Общие принципы ОС UNIX           | 9     |                                            | 16 |  | 27  | 52         |
| 2.    | Начала администрирования ОС UNIX | 9     |                                            | 20 |  | 27  | 56         |
|       | Итого:                           | 18    |                                            | 36 |  | 54  | 108        |

## Лабораторный практикум

| № п/п | № раз-дела дисциплины | Наименование лабораторных работ                                                          | Трудо-ём-кость (час.) |
|-------|-----------------------|------------------------------------------------------------------------------------------|-----------------------|
| 1.    | 1                     | Знакомство с операционной системой Linux                                                 | 4                     |
| 2.    | 1                     | Основы интерфейса взаимодействия пользователя с системой Unix на уровне командной строки | 2                     |
| 3.    | 1                     | Анализ файловой структуры UNIX. Команды для работы с файлами и каталогами                | 2                     |
| 4.    | 1                     | Поиск файлов. Перенаправление ввода-вывода.                                              | 2                     |
| 5.    | 1                     | Просмотр запущенных процессов                                                            | 2                     |
| 6.    | 1                     | Командная оболочка Midnight Commander                                                    | 2                     |
| 7.    | 1                     | Текстовый редактор vi                                                                    | 2                     |
|       |                       | Текстовый редактор emacs                                                                 | 2                     |
| 8.    | 2                     | Программирование в командном процессоре ОС UNIX. Командные файлы                         | 2                     |
| 9.    | 2                     | Программирование в командном процессоре ОС UNIX. Ветвления и циклы                       | 2                     |
| 10.   | 2                     | Программирование в командном процессоре ОС UNIX. Расширенное программирование            | 2                     |
| 11.   | 2                     | Средства для создания приложений в ОС UNIX                                               | 2                     |
| 12.   | 2                     | Управление версиями                                                                      | 2                     |
| 13.   | 2                     | Именованные каналы                                                                       | 4                     |
| 14.   | 2                     | Очереди сообщений                                                                        | 4                     |
| 15.   | 2                     | Сокеты                                                                                   | 4                     |
|       | Итого                 |                                                                                          | 36                    |

## Методические рекомендации по организации изучения дисциплины

### Структура практических (лабораторных) занятий

1. Задания по лабораторным работам выполняются малой группой (2-3 человека) в дисплейных классах в соответствии с календарным планом и методическими указаниями по выполнению лабораторных работ по дисциплине.
2. По результатам выполнения каждой лабораторной работы студентом готовится отчёт.

### Самостоятельная работа студента

1. Часть лабораторных работ предусматривает задания для индивидуальной самостоятельной работы студента, обязательные для выполнения.
2. Выполнение заданий для самостоятельной работы позволяет студенту приобрести дополнительные навыки и закрепить знания по изучаемой теме.



**Балльно-рейтинговая система оценки уровня знаний**  
**Сводная оценочная таблица дисциплины**

| Раздел                           | Тема                                 | Формы контроля уровня освоения ООП |               |                    |                                 | Баллы<br>темы | Баллы<br>раздела |
|----------------------------------|--------------------------------------|------------------------------------|---------------|--------------------|---------------------------------|---------------|------------------|
|                                  |                                      | Тесты к лекциям                    | Выполнение ЛР | Промежуточный тест | Итоговый контроль знаний (тест) |               |                  |
| Общие принципы ОС UNIX           | Введение в операционную систему UNIX | 1                                  | 0             | 3                  |                                 | 4             | 48               |
|                                  | Архитектура UNIX. Файлы и устройства | 1                                  | 8             | 3                  |                                 | 12            |                  |
|                                  | Архитектура UNIX. Процессы           | 1                                  | 12            | 3                  |                                 | 16            |                  |
|                                  | Терминал и командная строка          | 1                                  | 12            | 3                  |                                 | 16            |                  |
| Начала администрирования ОС UNIX | Введение в безопасность UNIX         | 1                                  | 12            |                    | 5                               | 18            | 52               |
|                                  | Сеть в UNIX                          | 1                                  | 8             |                    | 5                               | 14            |                  |
|                                  | Управление службами                  | 1                                  | 8             |                    | 5                               | 14            |                  |
|                                  | Управление программным обеспечением  | 1                                  | 0             |                    | 5                               | 6             |                  |
|                                  | Итого:                               | 8                                  | 60            | 12                 | 20                              | 100           | 100              |

**Таблица соответствия баллов и оценок**

| Баллы БРС | Традиционные оценки в РФ | Баллы для перевода оценок | Оценки | Оценки ECTS |
|-----------|--------------------------|---------------------------|--------|-------------|
| 86–100    | 5                        | 95–100                    | 5+     | A           |
|           |                          | 86–94                     | 5      | B           |
| 69–85     | 4                        | 69–85                     | 4      | C           |
| 51–68     | 3                        | 61–68                     | 3+     | D           |
|           |                          | 51–60                     | 3      | E           |
| 0–50      | 2                        | 31–50                     | 2+     | FX          |
|           |                          | 0–30                      | 2      | F           |

**Правила применения БРС**

1. Раздел (тема) учебной дисциплины считаются освоенными, если студент набрал более 50% от возможного числа баллов по этому разделу (теме).
2. Студент не может быть аттестован по дисциплине, если он не освоил все темы и разделы дисциплины, указанные в сводной оценочной таблице дисциплины.
3. По решению преподавателя и с согласия студентов, не освоивших отдельные разделы (темы) изучаемой дисциплины, в течение учебного семестра могут быть повторно проведены мероприятия текущего контроля успеваемости или выданы дополнительные учебные задания по этим темам или разделам. При этом студентам за данную работу засчитывается минимально возможный положительный балл (51 % от максимального балла).
4. При выполнении студентом дополнительных учебных заданий или повторного прохождения мероприятий текущего контроля полученные им баллы засчитываются за конкретные темы. Итоговая сумма баллов не может превышать максимального количества баллов, установленного по данным темам (в соответствии с приказом Ректора № 564 от 20.06.2013). По решению преподавателя предыдущие баллы, полученные студентом по учебным заданиям, могут быть аннулированы.
5. График проведения мероприятий текущего контроля успеваемости формируется в соответствии с календарным планом курса. Студенты обязаны сдавать все задания в сроки, установленные преподавателем.
6. Время, которое отводится студенту на выполнение мероприятий текущего контроля успеваемости, устанавливается преподавателем. По завершении отведённого времени студент должен сдать работу преподавателю, вне зависимости от того, завершена она или нет.
7. Использование источников (в том числе конспектов лекций и лабораторных работ) во время выполнения контрольных мероприятий возможно только с разрешения преподавателя.
8. Отсрочка в прохождении мероприятий текущего контроля успеваемости считается уважительной только в случае болезни студента, что подтверждается наличием у него медицинской справки, заверенной круглой печатью в поликлинике, предоставляемой преподавателю не позднее двух недель после выздоровления. В этом случае выполнение контрольных мероприятий осуществляется после выздоровления студента в срок, назначенный преподавателем. В противном случае отсутствие студента на контрольном мероприятии признается неуважительным.

9. При невыполнении контрольных заданий по лекциям студент не допускается к промежуточной контрольной работе.
10. Студент допускается к итоговому контролю знаний с любым количеством баллов, набранных в семестре.
11. Итоговая контроль знаний оценивается из 20 баллов независимо от числа баллов за семестр.
12. Если в итоге за семестр студент получил менее 31 балла, то ему выставляется оценка F и студент должен повторить эту дисциплину в установленном порядке. Если же в итоге студент получил 31–50 баллов, т. е. FX, то студенту разрешается добор необходимого (до 51) количества баллов путём повторного одноразового выполнения предусмотренных контрольных мероприятий, при этом по усмотрению преподавателя аннулируются соответствующие предыдущие результаты. Ликвидация задолженностей проводится в период с 07.02 по 28.02 (с 07.09 по 28.09) по согласованию с деканатом.

### **Учебно-методическое и информационное обеспечение дисциплины**

#### **а) Основная литература**

1. Робачевский А., Немнюгин С., Стесик О. Операционная система UNIX. — 2-е издание. — Санкт-Петербург: BHV, 2010.
2. Таненбаум А. Современные операционные системы. — Изд-во Питер, 2011.

#### **б) Дополнительная литература**

1. Ван Стеен М., Таненбаум Э. Распределенные системы. Принципы и парадигмы. — Изд-во: Питер, 2003.
2. Вудхалл А., Таненбаум Э. Операционные системы: разработка и реализация. Изд-во: Питер, 2006.
3. Немеет Э., Снайдер Г., Сибасс С, и др. Unix. Руководство системного администратора. — Киев: BHV, 1997.
4. Бек Л. Введение в системное программирование. — М. Мир, 1988.

#### **в) Программное обеспечение: ОС Linux, gcc, mc, vi, emacs, git.**

#### **г) Базы данных, информационно-справочные и поисковые системы: не рассмотрены.**

## Календарный план

Курс «Операционные системы», 1 курс, 2 семестр.

Трудоёмкость — 3 зачётные единицы, 18 недель, лекции — 1 час. в неделю, лабораторные работы — 2 час. в неделю.

### Виды и содержание учебных занятий

| Неделя | Лекции                               | Число часов | Лабораторные занятия       | Число часов |
|--------|--------------------------------------|-------------|----------------------------|-------------|
| 1      | Введение в операционную систему UNIX | 1           | Лабораторная работа № 1    | 2           |
| 2      | Введение в операционную систему UNIX | 1           | Лабораторная работа № 2    | 2           |
| 3      | Архитектура UNIX. Файлы и устройства | 1           | Лабораторная работа № 3    | 2           |
| 4      | Архитектура UNIX. Файлы и устройства | 1           | Лабораторная работа № 4    | 2           |
| 5      | Архитектура UNIX. Процессы           | 1           | Лабораторная работа № 5    | 2           |
| 6      | Архитектура UNIX. Процессы           | 1           | Лабораторная работа № 6    | 2           |
| 7      | Терминал и командная строка          | 1           | Лабораторная работа № 7    | 2           |
| 8      | Терминал и командная строка          | 1           | Лабораторная работа № 8    | 2           |
| 9      | Введение в безопасность UNIX         | 1           | Лабораторная работа № 9    | 2           |
| 10     | Промежуточная контрольная работа     | 1           | Лабораторная работа № 10   | 2           |
| 11     | Сеть в UNIX                          | 1           | Лабораторная работа № 11   | 2           |
| 12     | Сеть в UNIX                          | 1           | Лабораторная работа № 12   | 2           |
| 13     | Управление службами                  | 1           | Лабораторная работа № 13   | 2           |
| 14     | Управление службами                  | 1           | Лабораторная работа № 14   | 2           |
| 15     | Управление программным обеспечением  | 1           | Лабораторная работа № 15   | 2           |
| 16     | Управление программным обеспечением  | 1           | Досдача лабораторных работ | 2           |
| 17     | Заключительный обзор курса.          | 1           | Досдача лабораторных работ | 2           |
|        | Итоговая контрольная работа          | 1           | Консультация               | 2           |

## **Сведения об авторах**

Кулябов Дмитрий Сергеевич — доцент, кандидат физико-математических наук, доцент кафедры прикладной информатики и теории вероятностей РУДН.

Геворкян Мигран Нельсонович — кандидат физико-математических наук, доцент кафедры прикладной информатики и теории вероятностей РУДН.

Королькова Анна Владиславовна — доцент, кандидат физико-математических наук, доцент кафедры прикладной информатики и теории вероятностей РУДН.

Демидова Анастасия Вячеславовна — кандидат физико-математических наук, старший преподаватель кафедры прикладной информатики и теории вероятностей РУДН.

Учебное издание

**Дмитрий Сергеевич Кулябов, Мигран Нельсонович Геворкян,  
Анна Владиславовна Королькова, Анастасия Вячеславовна Демидова**

## **Операционные системы**

Редактор *И. Л. Панкратова*  
Технический редактор *Н. А. Ясько*  
Компьютерная вёрстка *А. В. Королькова, Д. С. Кулябов*

Подписано в печать 20.12.2016 г. Формат 60×84/16. Печать офсетная.  
Усл. печ. л. \_\_\_\_\_. Тираж 500 экз. Заказ № 1449.

---

Российский университет дружбы народов  
115419, ГСП-1, г. Москва, ул. Орджоникидзе, д. 3

---

Типография РУДН  
115419, ГСП-1, г. Москва, ул. Орджоникидзе, д. 3, тел. 952-04-41