МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

БАШКИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Салимоненко Д.А.

Методические указания по курсу: «Операционные системы»

Часть 1

Уфа 2014

Салимоненко Д.А. Реализация консольной версии системной оболочки: Методические указания /Изд-во Башкирск. ун-та.-Уфа, 2014.-76с.

Методические указания по реализации консольной версии системной оболочки, в первую очередь, предназначены для студентов кафедры «Программирование и Экономическая Информатика» Башкирского Государственного Университета. Излагаются ключевые аспекты создания программы для работы с файлами и каталогами, т.е. системной оболочки. Рассматривается алгоритмизация таких команд, как создание, копирование, переименование, удаление файлов и каталогов, а также проверка их свойств и значений.. Изложение методически продумано и рассчитано на использование при программирование на языке СИ++ под Linux.

Для адекватной работы с Методическими указаниями необходимы, по крайней мере, начальные познания в области программирования на языке СИ++. Но, они будут полезно и для тех, кто начинает работу с Си++ «с нуля». Студенты, уже имеющие достаточный опыт программирования на Си++, могут при изучении настоящих указаний пропустить части 1-3, а перейти сразу к части 4, где показывается, как создать простую консольную версию системной оболочки.

Издание 2-е. Исправлены замеченные опечатки и неточности.

Библ. 3 наимен.

Печатается по решению кафедры программирования и экономической информатики БашГУ.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Данные методические указания являются методическим материалом для студентов, изучающих предметы «операционные системы», «системное прикладное программирование» и аналогичные. В нем рассмотрены особенности создания системной оболочки (т.е. программы, позволяющей работать с файловой системой) в операционной системе Linux.

Первая часть посвящена общим вопросам программирования на Си++. В частности, затронуты вопросы мультифайлового программирования (создания программы из нескольких файлов), рассмотрена методика компиляции программ и их сборка в один (бинарный или двоичный) файл.

Во второй части показано, как создавать свои (авторские) библиотеки и подключать их к своим программам.

В третьей части рассмотрены вопросы реализации низкоуровневого ввода-вывода, на примерах разъяснено использование операторов работы с файлами, таких, как **open, read, write, close** и др.

В четвертой части приведен пример последовательной разработки системной оболочки. Приведены готовые коды программ.

Оболочка операционной системы (от англ. shell— оболочка) — интерпретатор команд операционной системы (ОС), обеспечивающий интерфейс для взаимодействия пользователя с функциями системы.

В общем случае, различают оболочки с двумя типами интерфейса для взаимодействия с пользователем: текстовый пользовательский интерфейс (CUI) и графический пользовательский интерфейс (GUI).

Командный интерпретатор

Для обеспечения интерфейса командной строки в ОС часто используются командные интерпретаторы, которые могут

представлять собой самостоятельные языки программирования, с собственным синтаксисом и отличительными функциональными возможностями.

В операционные системы MS-DOS и Windows 9x включён командный интерпретатор command.com, в Windows NT включён cmd.exe. В большом семействе командных оболочек UNIX популярны bash, csh, ksh, zsh и другие.

Как правило, при низкоуровневой настройке ОС у пользователя есть возможность менять командный интерпретатор, используемый по умолчанию.

Функции

Командный интерпретатор исполняет команды своего языка, заданные в командной строке или поступающие из стандартного ввода или указанного файла.

В качестве команд интерпретируются вызовы системных или прикладных утилит, а также управляющие конструкции. Кроме того, оболочка отвечает за раскрытие шаблонов имен файлов и за перенаправление и связывание ввода-вывода утилит.

В совокупности с набором утилит, оболочка представляет собой операционную среду, полноценный язык программирования и мощное средство решения как системных, так и некоторых прикладных задач, в особенности, автоматизации часто выполняемых последовательностей команд.

Стандартный командный интерпретатор

Стандартом POSIX (ISO/IEC 9945) (Том 3. Оболочка и утилиты) определен язык оболочки, включающий конструкции последовательного (перевод строки, точка с запятой), условного («if», «case», «||», «&&») и циклического («for», «for in», «while», «until») исполнения команд, а также оператор присваивания.

Стандартом также определен режим редактирования вводимых команд, являющийся подмножеством команд стандартного текстового редактора («vi»).

В современных открытых Unix-подобных ОС наиболее распространены такие языки командных интерпретаторов как bash и zsh, реализующие надмножества языка стандартной оболочки.

Для альтернативных ОС (например, Windows) также доступны реализации этих оболочек.

Альтернативы

Наряду со стандартными, в открытых ОС применяются также альтернативные оболочки csh и tcsh, отличающиеся синтаксисом управляющих конструкций и поведением переменных.

Некоторые альтернативные ОС поставляются с интерпретаторами собственных языков командных файлов (такими, как язык командных файлов ОС MS-DOS и Microsoft Windows 7, язык kazahskii в ОС OS/2 и т. п.)

Некоторые предпочитают пользоваться для автоматизации часто выполняемых последовательностей команд новыми интерпретируемыми языками, например, Perl или Python.

Графическая оболочка

Последние версии ОС Windows используют в качестве своей оболочки интегрированную среду Проводника Windows. Проводник Windows представляет собой визуальную среду управления включающую в себя Рабочий стол, Меню пуск, Панель задач, а также функции управления файлами. Ранние версии ОС Windows 3.xx в качестве графической оболочки включают менеджер программ.

Многие сторонние разработчики предлагают альтернативные среды, которые могут быть использованы вместо оболочки проводника, включенной по умолчанию компанией Microsoft в систему Windows.

Перечень оболочек для Microsoft Windows:

- Aston shell
- BB4Win
- BBIean
- Cairo (Under Development)
- Chroma
- Emerge Desktop
- Geoshell
- KDE

• Litestep
• Packard Bell Navigator
• Program Manager
• Secure Desktop
• SharpE
• Talisman Desktop
• WinStep
• Windows Explorer
• Microsoft Bob

Далее, мы будем рассматривать системную оболочку типа CUI, то есть текстовый пользовательский интерфейс. Рассмотрим цели, которые программа должна будет выполнять:

• Предоставляет возможность создания файла
• Предоставляет возможность удаления файла
• Предоставляет возможность переименование файла
• Предоставляет возможность копировать файл
• Предоставляет возможность получить информацию о файле или каталоге
• Предоставляет возможность создания каталога
• Предоставляет возможность удаления каталога (с вложенными каталогами и файлами)
• Работа в операционной системе Linux

Данная системная оболочка не отличается функциональностью, но мы и не стремимся превзойти готовые продукты. Наша цель- понять устройство системной оболочки на примере её создания.

При создании системной оболочки будет пользоваться модульным подходом. То есть создадим отдельно модули для каждой задачи, а потом будем вызывать их из основного модуля. Это позволит добавлять новые модули в нашу программу практически ничего не меняя. Кроме того, такой подход позволяет локализировать ошибки. Список модулей:

• Создание или открытие файла
• Вывод ошибки
• Удаление файла

• Переименование файла
• Копирование файла
• Вывод информации о файле или каталоге
• Создание каталога
• Удаление каталога
• Основной модуль

При написании системной оболочки можно пользоваться языком программирования C/C++ и, например, средой разработки IDE NetBeans.

1. Вводная часть. Мультифайловое программирование, сборка программы

Материалы для данной части подобраны на основе информации с сайта http://www.linuxcenter.ru/

1.1. Как компилировать программу

Чтобы сразу начать программировать, создадим еще один клон известной программы "Hello World". Что делает эта программа, вы знаете. Откройте свой любимый текстовый редактор и наберите в нем следующий текст:

```
/* hello.c */
#include <stdio.h>
int main (void)
{
        printf ("Hello World\n");
}
```

Я назвал свой файл **hello.c**. Вы можете назвать как угодно, сохранив суффикс .**c**.

Содержимое файла **hello.c** - это исходный код программы ('**program, source**', '**source code**' или просто '**source**'). А **hello.c** - это исходный файл программы ('**source file**'). Hello World - очень маленькая программа, исходный код которой помещается в одном файле. В "настоящих" программах, как правило, исходный код разносится по нескольким файлам. В больших программах исходных файлов может быть больше сотни.

Наш исходный код написан на языке программирования С. Языки программирования были придуманы для того, чтобы программист мог объяснить компьютеру, что делать. Но вот беда, компьютер не понимает ни одного языка программирования. У компьютера есть свой язык, который называют машинным кодом или исполняемым кодом ('executable code'). Написать Hello World в

машинном коде можно, но серьезные программы на нем не пишутся. Исполняемый код не только сложный по своей сути, но и очень неудобный для человека. Программа, которую можно написать за один день на языке программирования будет писаться целый год в машинном коде. Потом программист сойдет с ума. Чтобы этого не случилось, был придуман компилятор ('compiler'), который переводит исходный код программы в исполняемый код. Процесс перевода исходного кода программы в исполняемый код называют компиляцией.

Чтобы откомпилировать наш Hello World достаточно набрать в командной строке следующее заклинание:

```
$ gcc -o hello hello.c
$
```

Если исходный код написан без синтаксических ошибок, то компилятор завершит свою работу без каких-либо сообщений. Молчание - знак повиновения и согласия. Набрав команду **ls** вы тут же обнаружите новый файл с именем **hello.** Этот файл содержит исполняемый код программы. Такие файлы называют исполняемыми файлами ('**executable files**') или бинарниками ('**binary files**').

Вы наверняка догадались, что опция **-o** компилятора **gcc** указывает на то, каким должно быть имя выходного файла. Как вы позже узнаете, выходным файлом может быть не только бинарник. Если не указать опцию **-o**, то бинарнику, в нашем случае, будет присвоено имя **a.out.**

Осталось только запустить полученный бинарник. Для этого набираем в командной строке следующую команду:

```
$ ./hello
Hello World
$
```

Когда мы набираем в командной строке путь к бинарнику, мы, в реальности сообщаем оболочке, что надо выполнить программу. Оболочка "передает" бинарник ядру операционной системе, а ядро системы особым шаманским способом отдает программу на выполнение процессору. Затем, если программа не

была запущена в фоновом режиме, то оболочка ждет от ядра сообщения о том, что программа выполнилась. Получив такое сообщение, оболочка выдает приглашение на ввод новой команды. Вы можете еще раз набрать **./hello** и процедура повторится. В нашем случае программа выполняется очень быстро, и новое приглашение командной строки "вылетает" практически сразу.

Мы рассмотрели идеальный случай, когда программа написана без синтаксических ошибок. Попробуем намеренно испортить программу таким образом, чтобы она не отвечала канонам языка С. Для этого достаточно убрать точку с запятой в конце вызова функции **printf()**:

printf ("Hello World\n")

Теперь, если попытаться откомпилировать программу, то компилятор выругается, указав нам на то, что он считает неправильным:

```
$ gcc -o hello hello.c
hello.c: In function 'main':
hello.c:7: error: syntax error before '}' token
$
```

В первой строке говорится, что в файле **hello.c** (у нас он единственный) в теле функции **main()** что-то произошло. Вторая строка сообщает, что именно произошло: седьмая строка файла **hello.c** вызвала ошибку (error). Далее идет расшифровка: синтаксическая ошибка перед закрывающейся фигурной скобкой.

Заглянув в файл **hello.c** мы с удивлением обнаружим, что нахулиганили мы не в седьмой, а в шестой строке. Дело в том, что компилятор обнаружил нелады только в седьмой строке, но написал 'before' (до), что означает "прокручивай назад".

Естественно, пока мы не исправим ошибку, ни о каком бинарнике не может идти и речи. Если мы удалим старый бинарник **hello**, доставшийся нам от прошлой компиляции, то увидим, что компиляция испорченного файла не даст никакого результата. Однако иногда компилятор может лишь "заподозрить" что-то неладное, потенциально опасное для нормального существования программы. Тогда вместо 'error' пишется 'warning'

(предупреждение), и бинарник все-таки появляется на свет (если в другом месте нет явных ошибок). Не следует игнорировать предупреждения, за исключением тех случаев, когда вы на 100% знаете, что делаете.

Парадокс программирования заключается в том, что можно наделать кучу ошибок (уже не синтаксических, как в нашем случае, а смысловых) по всем правилам языка программирования. В таком случае компилятор выдает бинарник, который делает не то, что мы хотели. В таком случае программу приходится отлаживать. Отладка - это обычное дело при написании любой достаточно сложной программы. Не ошибается только тот, кто ничего не делает.

1.2. Мультифайловое программирование

Как уже говорилось, если исходный код сколько-нибудь серьезной программы уместить в одном файле, то такой код станет просто нечитаемым. К тому же если программа компилируется достаточно долго (особенно это относится к языку С++), то после исправления одной ошибки, нужно перекомпилировать весь код.

Куда лучше разбросать исходный код по нескольким файлам (осмысленно, по какому-нибудь критерию), и компилировать каждый такой файл отдельно. Как вы вскоре узнаете, это очень даже просто.

Давайте сначала разберемся, как из исходного файла получается бинарник. Подобно тому как гусеница не сразу превращается в бабочку, так и исходный файл не сразу превращается в бинарник. После компиляции создается объектный код. Это исполняемый код с некоторыми "вкраплениями", из-за которых объектный код еще не способен к выполнению. Сразу в голову приходит стиральная машина: вы ее только что купили и она стоит у вас дома в коробке. В таком состоянии она стирать не будет, но вы все равно рады, потому что осталось только вытащить из коробки и подключить.

Вернемся к объектному коду. Эти самые "вкрапления" (самое главное среди них - таблица символов) позволяют объектному коду "пристыковываться" к другому объектному коду. Такой фокус делает компоновщик (линковщик) - программа, которая объединяет объектный код, полученный из "разных мест", удаляет все лишнее и создает полноценный бинарник. Этот процесс называется компоновкой или линковкой.

Итак, чтобы откомпилировать мультифайловую программу, надо сначала добыть объектный код из каждого исходного файла в отдельности. Каждый такой код будет представлять собой объектный модуль. Каждый объектный модуль записывается в отдельный объектный файл. Затем объектные модули надо скомпоновать в один бинарник.

В Linux в качестве линковщика используется программа **ld**, обладающая приличным арсеналом опций. К счастью **gcc** самостоятельно вызывает компоновщик с нужными опциями, избавляя нас от "ручной" линковки.

Попробуем теперь, вооружившись запасом знаний, написать мультифайловый Hello World. Создадим первый файл с именем **main.c**:

```
/* main.c */
int main (void)
{
        print_hello ();
}
```

Теперь создадим еще один файл **hello.c** со следующим содержимым:

```
/* hello.c */
#include <stdio.h>
void print_hello (void)
{
        printf ("Hello World\n");
}
```

Здесь функция **main()** вызывает функцию **print_hello()**, находящуюся в другом файле. Функция **print_hello()** выводит на

экран заветное приветствие. Теперь нужно получить два объектных файла. Опция **-c** компилятора **gcc** заставляет его отказаться от линковки после компиляции. Если не указывать опцию **-o**, то в имени объектного файла расширение .c будет заменено на .o (обычные объектные файлы имеют расширение **.o**):

```
$ gcc -c main.c
$ gcc -c hello.c
$ls
hello.c hello, o main.c main. o
$
```

Итак, мы получили два объектных файла. Теперь их надо объединить в один бинарник:

```
$ gcc -o hello main.o hello.o
$ls
hello* hello.c hello.o main.c. main.o
$ ./hello
Hello World
$
```

Компилятор "увидел", что вместо исходных файлов (с расширением **.c**) ему подбросили объектные файлы (с расширением **.o**) и отреагировал согласно ситуации: вызвал линковщик с нужными опциями.

Давайте разберемся, что же все-таки произошло. В этом нам поможет утилита **nm**. Выше говорилось, что объектные файлы содержат таблицу символов. Утилита **nm** как раз позволяет посмотреть эту таблицу в читаемом виде. Те, кто пробовал программировать на ассемблере знают, что в исполняемом файле буквально все (функции, переменные) стоит на своей позиции: стоит только вставить или убрать из программы один байт, как программа тут же превратиться в груду мусора из-за смещенных позиций (адресов). У объектных файлов особая роль: они хранят в таблице символов имена некоторых позиций (глобально объявленных функций, например). В процессе линковки происходит стыковка имен и пересчет позиций, что позволяет нескольким объектным файлам объединиться в один бинарник.

Если вызвать **nm** для файла **hello.o**, то увидим следующую картину:

$ nm hello.o
U printf
00000000 T print_hello
$

Отложим разговор о смысловой нагрузке нулей и литер U, T. Важным является то, что в объектном файле сохранилась информация об использованных именах. Своя информация есть и в файле **main.o**:

$ nm main.o
00000000 T main U print_hello
$

Таблицы символов объектных файлов содержат общее имя **print_hello**. В процессе линковки высчитываются и подставляются в нужные места адреса, соответствующие именам из таблицы. Вот и весь секрет.

1.3. Автоматическая сборка

В предыдущем разделе для создания бинарника из двух исходных файлов нам пришлось набрать три команды. Если бы программу пришлось отлаживать, то каждый раз надо было бы вводить одни и те же три команды. Казалось бы, выход есть: написать сценарий оболочки. Но давайте подумаем, какие в этом случае могут быть недостатки. Во-первых, каждый раз сценарий будет компилировать все файлы проекта, даже если мы исправили только один из них. В нашем случае это не страшно. Но если речь идет о десятках файлов! Во-вторых, сценарий "намертво" привязан к конкретной оболочке. Программа тут же становится менее переносимой. И, наконец, простому скрипту не хватает функциональности (задание аргументов сборки и т. п.), а хороший скрипт (с многофункциональными прибамбасами) плохо модернизируется.

Выход из сложившейся ситуации есть. Это утилита make, которая работает со своими собственными сценариями. Сценарий записывается в файле с именем **Makefile** и помещается в репозиторий (рабочий каталог) проекта. Сценарии утилиты make просты и многофункциональны, а формат **Makefile** используется повсеместно (и не только на Unix-системах). Дошло до того, что стали создавать программы, генерирующие **Makefile'ы**. Самый яркий пример - набор утилит GNU Autotools. Самое главное преимущество **make** - это "интеллектуальный" способ рекомпиляции: в процессе отладки **make** компилирует только измененные файлы.

То, что выполняет утилита **make**, называется сборкой проекта, а сама утилита make относится к разряду сборщиков.

Любой **Makefile** состоит из трех элементов: комментарии, макроопределения и целевые связки (или просто связки). В свою очередь связки состоят тоже из трех элементов: цель, зависимости и правила.

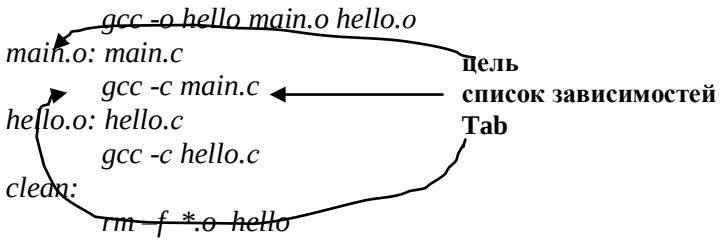
Сценарии make используют однострочные комментарии, начинающиеся с литеры # (решетка). О том, что такое комментарии и зачем они нужны, объяснять не буду.

Макроопределения позволяют назначить имя практически любой строке, а затем подставлять это имя в любое место сценария, где должна использоваться данная строка. Макросы **Makefile** схожи с макроконстантами языка C.

Связки определяют: 1) что нужно сделать (цель); 2) что для этого нужно (зависимости); 3) как это сделать (правила). В качестве цели выступает имя или макроконстанта. Зависимости - это список файлов и целей, разделенных пробелом. Правила - это команды передаваемые оболочке.

Теперь рассмотрим пример. Попробуем составить сценарий сборки для рассмотренного в предыдущем разделе мультифайлового проекта Hello World. Создайте файл с именем **Makefile**:

 # Makefile for Hello World project
hello: main.o hello.o



Обратите внимание, что в каждой строке перед вызовом **gcc**, а также в строке перед вызовом **rm** стоят табуляции. Как вы уже догадались, эти строки являются правилами. Формат **Makefile** требует, чтобы каждое правило начиналось с табуляции. Теперь рассмотрим все по порядку.

Makefile может начинаться как с заглавной так и со строчной буквы. Но рекомендуется все-таки начинать с заглавной, чтобы он не перемешивался с другими файлами проекта, а стоял "в списке первых".

Первая строка - комментарий. Здесь можно писать все, что угодно. Комментарий начинается с символа # (решетка) и заканчивается символом новой строки. Далее по порядку следуют четыре связки: 1) связка для компоновки объектных файлов **main.o** и **hello.o**; 2) связка для компиляции **main.c**; 3) связка для компиляции **hello.c**; 4) связка для очистки проекта.

Первая связка имеет цель hello. Цель отделяется от списка зависимостей двоеточием. Список зависимостей отделяется от правил символом новой строки. А каждое правило начинается на новой строке с символа табуляции. В нашем случае каждая связка содержит по одному правилу. В списке зависимостей перечисляются через пробел вещи, необходимые для выполнения правила. В первом случае, чтобы скомпоновать бинарник, нужно иметь два объектных файла, поэтому они оказываются в списке зависимостей. Изначально объектные файлы отсутствуют, поэтому требуется создать целевые связки для их получения. Итак, чтобы получить **main.o**, нужно откомпилировать **main.c**. Таким образом файл **main.c** появляется в списке зависимостей (он там единственный). Аналогичная ситуация с **hello.o**. Файлы **main.c** и

hello.c изначально существуют (мы их сами создали), поэтому никаких связок для их создания не требуется.

Особую роль играет целевая связка **clean** с пустым списком зависимостей. Эта связка очищает проект от всех автоматически созданных файлов. В нашем случае удаляются файлы **main.o, hello.o** и **hello**. Очистка проекта бывает нужна в нескольких случаях: 1) для очистки готового проекта от всего лишнего; 2) для пересборки проекта (когда в проект добавляются новые файлы или когда изменяется сам **Makefile**; 3) в любых других случаях, когда требуется полная пересборка (например, для измерения времени полной сборки).

Теперь осталось запустить сценарий. Формат запуска утилиты make следующий:

make [опции] [цели...]

Опции make нам пока не нужны. Если вызвать **make** без указания целей, то будет выполнена первая попавшаяся связка (со всеми зависимостями) и сборка завершится. Нам это и требуется:

```
$ make
gcc -c main.c
gcc -c hello.c
gcc -o hello main.o hello.o
$ls
hello* hello.c hello.o main.c main.o Makefile
$ ./hello
Hello World
$
```

В процессе сборки утилита **make** пишет все выполняемые правила. Проект собран, все работает.

Теперь давайте немного модернизируем наш проект. Добавим одну строку в файл **hello.c**:

```
/* hello.c */
#include <stdio.h>
void print_hello (void)
{
        printf ("Hello World\n");
```

```
        printf ("Goodbye World|n");
}
```

Теперь повторим сборку:

```
$ make
gcc -c hello.c
gcc -o hello main.o hello.o
$ ./hello
Hello World
Goodbye World
$
```

Утилита **make** "пронюхала", что был изменен только **hello.c**, то есть компилировать нужно только его. Файл **main.o** остался без изменений. Теперь давайте очистим проект, оставив одни исходники:

```
$ make clean
rm –f   *.o hello
$ls
hello.c main.c Makefile
$
```

В данном случае мы указали цель непосредственно в командной строке. Так как целевая связка **clean** содержит пустой список зависимостей, то выполняется только одно правило. Не забывайте "чистить" проект каждый раз, когда изменяется список исходных файлов или когда изменяется сам **Makefile.**

1.4. Модель КИС

Любая программа имеет свой репозиторий - рабочий каталог, в котором находятся исходники, сценарии сборки (**Makefile**) и прочие файлы, относящиеся к проекту. Репозиторий рассмотренного нами проекта мультифайлового **Hello World** изначально состоит из файлов **main.c, hello.c** и, собственно. **Makefile**. После сборки репозиторий дополняется файлами **main.o, hello.o и hello**. Практика показывает, что правильная организация

исходного кода в репозиторий не только упрощает модернизацию и отладку, но и предотвращает возможность появления многих ошибок.

Модель КИС (Клиент-Интерфейс-Сервер) - это элегантная концепция распределения исходного кода в репозиторий, в рамках которой все исходники можно поделить на клиенты, интерфейсы и серверы.

Итак, сервер предоставляет услуги. В нашем случае это могут быть функции, структуры, перечисления, константы, глобальные переменные и проч. В языке C++ это чаще всего классы или иерархии классов. Любой желающий (клиент) может воспользоваться предоставленными услугами, то есть вызвать функцию со своими фактическими параметрами, создать экземпляр структуры, воспользоваться константой и т.п. В C++, как правило, клиент использует класс как тип данных и использует его члены.

Часто бывает, что клиент сам становится сервером, точнее начинает играть роль. промежуточного сервера. Хороший пример - наш мультифайловый Hello World. Здесь функция **print_hello()** (клиент) пользуется услугами стандартной библиотеки языка C (сервер), вызывая функцию **printf()**. Однако в дальнейшем функция **print_hello()** сама становится сервером, предоставляя свои услуги функции **main()**. В языке C++ довольно часто клиент создает производный класс, который наследует некоторые механизмы базового класса сервера. Таким образом, клиент сам становится сервером, предоставляя услуги своего производного класса.

Клиент с сервером должны "понимать" друг друга, иначе взаимодействие невозможно. Интерфейс (протокол) - это условный набор правил, согласно которым взаимодействуют клиент и сервер. В нашем случае (мультифайловый Hello World) интерфейсом (протоколом) является общее имя в таблице символов двух объектных файлов. Такой способ взаимодействия может привести к неприятным последствиям. Клиент (функция **main()**) не знает ничего, кроме имени функции **print_hello()** и, наугад вызывает ее без аргументов и без присваивания. Иначе говоря, клиент не знает до конца правил игры. В нашем случае прототип функции

print_hello() неизвестен.

Обычно для организации интерфейсов используются объявления (прототипы), которые помещаются чаще всего в заголовочные файлы. В языке С это файлы с расширением .h; в языке С++ это файлы с раширением **.h, .hpp** или без расширения. Некоторые "всезнайки" ошибочно называют заголовочные файлы библиотеками и умудряются учить этому других. Забегая вперед отметим, что библиотека - это просто коллекция скомпонованных особым образом объектных файлов, а заголовочный файл - это интерфейс. Основная разница между библиотеками и заголовочньми файлами в том, что библиотека - это объектный (почти исполняемый) код, а заголовочный файл -это исходный код. Включая в программу заголовочный файл директивой **#include** мы соглашаемся работать с сервером (будь то библиотека или простой объектный файл) по его протоколу: если сервер сказал, что функция вызывается без аргументов, то она и будет вызываться без аргументов, иначе компилятор костьми ляжет, но не даст откомпилировать "незаконный вызов".

Вернемся к Hello World. В таком виде, как он есть сейчас, мы можем, например, вызвать функцию **print_hello()** с аргументом, и компилятор даже не заподозрит неладное, потому что на уровне исходного кода нет четких правил, регламентирующих взаимодействие клиента и сервера. После того как мы создадим заголовочный файл и включим его в файл **main.c**, компилятор будет "сматывать удочки" каждый раз, когда мы будем пытаться вызвать функцию **print_hello()** не по правилам. Таким образом интерфейс (набор объявлений, в данном случае - в заголовочном файле) - это публичная оферта сервера клиенту. Включение заголовочного файла директивой **#include** - это акцепт или подпись.

Еще хочу сказать пару слов о стандартной библиотеке языка С. Как уже отмечалось, библиотека - это набор объектных файлов, которые подсоединяются к программе на стадии линковки. Так как стандартная библиотека языка С - это часть стандарта языка С, то она подключается автоматически во время линковки

программы, но так как компилятор **gcc** сам вызывает линковщик с нужными параметрами, то мы этого просто не замечаем. Включая в исходники заголовочный файл **stdio.h**, мы автоматически соглашаемся использовать механизм стандартного ввода-вывода на условиях сервера (стандартной библиотеки языка С).

Теперь попробуем применить модель КИС на практике для нашего проекта Hello World. Создадим файл **hello, h**:

```
/* hello.h */
void print_hello (void);
```

Теперь включим этот файл в main.c:

```
/* main.c */
#include "hello.h"
int main (void)
{
        print_hello ();
}
```

Так как в проект был добавлен новый файл, надо сделать полную пересборку:

```
$ make clean
rm  -f  *.o  hello
$ make
gcc -c main.c
gcc -c hello.c
gcc -o hello main.o hello.o
$ ./hello
Hello World
Goodbye World
$
```

С виду ничего не изменилось. Но на самом деле программа стала более правильной, более изящной и, самое главное, более безопасной.

2. Библиотеки

2.1. Введение в библиотеки

Как уже неоднократно упоминалось в предыдущей главе, библиотека - это набор скомпонованных особым образом объектных файлов. Библиотеки подключаются к основной программе во время линковки. По способу компоновки библиотеки подразделяют на архивы (статические библиотеки, **static libraries**) и совместно используемые (динамические библиотеки, **shared libraries**). В Linux, кроме того, есть механизмы динамической подгрузки библиотек. Суть динамической подгрузки состоит в том, что запущенная программа может по собственному усмотрению подключить к себе какую-либо библиотеку. Благодаря этой возможности создаются программы с подключаемыми плагинами, такие как XMMS. В этой главе мы не будем рассматривать динамическую подгрузку, а остановимся на классическом использовании статических и динамических библиотек.

С точки зрения модели КИС, библиотека - это сервер. Библиотеки несут в себе одну важную мысль: возможность использовать одни и те же механизмы в разных программах. В Linux библиотеки используются повсеместно, поскольку это очень удобный способ "не изобретать велосипеды". Даже ядро Linux в каком-то смысле представляет собой библиотеку механизмов, называемых системными вызовами.

Статическая библиотека - это просто архив объектных файлов, который подключается к программе во время линковки. Эффект такой же, как если бы вы подключали каждый из файлов отдельно.

В отличие от статических библиотек, код совместно используемых (динамических) библиотек не включается в бинарник. Вместо этого в бинарник включается только ссылка на библиотеку.

Рассмотрим преимущества и недостатки статических и

совместно используемых библиотек. Статические библиотеки делают программу более автономной: программа, скомпонованная со статической библиотекой может запускаться на любом компьютере, не требуя наличия этой библиотеки (она уже "внутри" бинарника). Программа, скомпонованная с динамической библиотекой, требует наличия этой библиотеки на том компьютере, где она запускается, поскольку в бинарнике не код, а ссылка на код библиотеки. Не смотря на такую зависимость, динамические библиотеки обладают двумя существенными преимуществами. Во-первых, бинарник, скомпонованный с совместно используемой библиотекой меньше размером, чем такой же бинарник, с подключенной к нему статической библиотекой (статически скомпонованный бинарник). Во-вторых, любая модернизация динамической библиотеки, отражается на всех программах, использующих ее. Таким образом, если некоторую библиотеку **foo** используют 10 программ, то исправление какой-нибудь ошибки в **foo** или любое другое улучшение библиотеки автоматически улучшает все программы, которые используют эту библиотеку. Именно поэтому динамические библиотеки называют совместно используемыми. Чтобы применить изменения, внесенные в статическую библиотеку, нужно пересобрать все 10 программ.

В Linux статические библиотеки обычно имеют расширение **.a** (Archive), а совместно используемые библиотеки имеют расширение **.so** (Shared Object). Хранятся библиотеки, как правило, в каталогах **/lib** и **/usr/lib**. В случае иного расположения (относится только к совместно используемым библиотекам), приходится немного "подшаманить", чтобы программа запустилась.

2.2. Пример статической библиотеки

Теперь давайте создадим свою собственную библиотеку, располагающую двумя функциями:

h_world() и **g_world()**, которые выводят на экран "Hello World" и "Goodbye World" соответственно. Начнем со статической библиотеки.

Начнем с интерфейса. Создадим файл **world.h**:

```
/* world.h */
void h_world (void);
void g_world (void);
```

Здесь просто объявлены функции, которые будут использоваться.

Теперь надо реализовать серверы. Создадим файл **h_world.c**:

```
/* h world.c */
#include <stdio.h>
#include "world.h"
void h_world (void)
{
        printf ("Hello World\n");
}
```

Теперь создадим файл **g_world.c**, содержащий реализацию функции **g_world()**:

```
/* g_world.c */
#include <stdio.h>
#include "world.h"
void g_world (void)
{
        printf ("Goodbye World\n");
}
```

Можно было бы с таким же успехом уместить обе функции в одном файле (**hello.c** например), однако для наглядности мы разнесли код на два файла.

Теперь создадим файл **main.c**. Это клиент, который будет пользоваться услугами сервера:

```
/* main.c */
#include "world.h"
int main (void)
```



```
{
        h_world ();
        g_world ();
}
```

Теперь напишем сценарий для make. Для этого создаем **Makefile**:

```
# Makefile for World project
binary: main.o libworld.a
        gcc -o binary main.o -L. –I  world
main.o: main.c
        gcc -c main.c
libworld.a: h_world.o g_world.o
        ar cr libworld.a h_world.o g_world.o
h_world.o: h_world.c
        gcc -c h_world.c
g_world.o: g_world.c
        gcc -c g_world.c
clean:
        rm –f  *.o  *.a binary
```

Не забывайте ставить табуляции перед каждым правилом в целевых связках.

Собираем программу:

```
$ make
gcc -c main.c
gcc -c h_world.c
gcc -c g_world.c
ar cr libworld.a h_world.o g_world.o
gcc -o binary main.o -L. –I  world
```

Осталось только проверить, работает ли программа и разобраться, что же мы такое сделали:

```
$ ./binary Hello World
Goodbye World
$
```

Итак, в приведенном примере появились три новые вещи: опции **-I** и **-L** компилятора, а также команда **ar**. Начнем с

последней. Как вы уже догадались, команда **ar** создает статическую библиотеку (архив). В нашем случае два объектных файла объединяются в один файл **libworld.a**. В Linux практически все библиотеки имеют префикс **lib**.

Как уже говорилось, компилятор **gcc** сам вызывает линковщик, когда это нужно. Опция **-I**, переданная компилятору, обрабатывается и посылается линковщику для того, чтобы тот подключил к бинарнику библиотеку. Как вы уже заметили, у имени библиотеки "обрублены" префикс и суффикс. Это делается для того, чтобы создать "видимое безразличие" между статическими и динамическими библиотеками. Но об этом речь пойдет в других главах книги. Сейчас важно знать лишь то, что и библиотека **libfoo.so** и библиотека **libfoo.a** подключаются к проекту опцией **-lfoo**. В нашем случае **libworld.a** "урезалось" до **-world**.

Опция **-L** указывает линковщику, где ему искать библиотеку. В случае, если библиотека располагается в каталоге **/lib** или **/usr/lib**, то вопрос отпадает сам собой и опция **-L** не требуется. В нашем случае библиотека находится в репозитории (в текущем каталоге). По умолчанию линковщик не просматривает текущий каталог в поиске библиотеки, поэтому опция **-L**. (точка означает текущий каталог) необходима.

2.3. Пример совместно используемой библиотеки

Для того, чтобы создать и использовать динамическую (совместно используемую) библиотеку, достаточно переделать в нашем проекте **Makefile**.

```
# Makefile for World project
binary: main.o libworld.so
        gcc -o binary main.o -L. –I  world –Wl, -rpath,.
main.o: main.c
        gcc -c main.c
libworld.so: h_world.o g_world.o
        gcc -shared -o libworld.so h_world.o g_world.o
```



```
h_world.o: h_world.c
        gcc -c -fPIC h_world.c
g_world.o: g_world.c
        gcc -c -fPIC  g_world.c
        clean:
        rm  –f  *.o  *.so binary
```

Внешне ничего не изменилось: программа компилируется, запускается и выполняет те же самые действия, что и в предыдущем случае. Изменилась внутренняя суть, которая играет для программиста первоочередную роль. Рассмотрим все по порядку.

Правило для сборки binary теперь содержит пугающую опцию **-Wl**, **-rpath**. Ничего страшного тут нет. Как уже неоднократно говорилось, компилятор gcc сам вызывает линковщик **ld**, когда это надо и передает ему нужные параметры сборки, избавляя нас от ненужной платформенно-зависимой волокиты. Но иногда мы все-таки должны вмешаться в этот процесс и передать линковщику "свою" опцию. Для этого используется опция компилятора ***-Wl,* option, optargs,...**

Расшифровываю: передать линковщику (**-Wl**) опцию **option** с аргументами **optargs**. В нашем случае мы передаем линковщику опцию -rpath с аргументом . (точка, текущий-каталог). Возникает вопрос: что означает опция **-rpath**? Как уже говорилось, линковщик ищет библиотеки в определенных местах; обычно это каталоги **/lib** и **/usr/lib**, иногда **/usr/local/lib**. Опция **-rpath** просто добавляет к этому списку еще один каталог. В нашем случае это текущий каталог. Без указания опции **-rpath**, линковщик "молча" соберет программу, но при запуске нас будет ждать сюрприз: программа не запустится из-за отсутствия библиотеки. Попробуйте убрать опцию ***-Wl,* -rpath**. из **Makefile** и пересоберите проект. При попытке запуска программа **binary** завершится с кодом возврата 127. То же самое произойдет, если вызвать программу из другого каталога. Верните обратно ***-Wl,* -rpath,.**, пересоберите проект, поднимитесь на уровень выше командой **cd..** и попробуйте запустить бинарник командой **world/binary**. Ничего не получится,

поскольку в новом текущем каталоге библиотеки нет.

Есть один способ не передавать линковщику дополнительных опций при помощи **-Wl** - это использование переменной окружения **LD_LIBRARY_PATH**.. Сейчас лишь скажу, что у каждого пользователя есть так называемое окружение (environment) представляющее собой набор пар ПЕРЕМЕННАЯ=ЗНАЧЕНИЕ, используемых программами. Чтобы посмотреть окружение, достаточно набрать команду **env.** Чтобы добавить в окружение переменную, достаточно набрать export ПЕРЕМЕННАЯ=ЗНАЧЕНИЕ, а чтобы удалить переменную из окружения, надо набрать export **-n** ПЕРЕМЕННАЯ. Будьте внимательны: export - это внутреннаяя команда оболочки BASH; в других оболочках (csh, ksh, ...) используются другие команды для работы с окружением. Переменная окружения **LD_LIBRARY_PATH** содержит список дополнительных "мест", разделенных двоеточиеями, где линковщих должен искать библиотеку.

Не смотря на наличие двух механизмов передачи информации о нестандартном расположении библиотек, лучше помещать библиотеки в конечных проектах в **/lib** и в **/usr/lib.** Допускается расположение библиотек в подкаталоги **/usr/lib** и в **/usr/local/lib** (с указанем **-Wl**, **-rpath**). Но заставлять конечного пользователя устанавливать **LD_LIBRARY_PATH** почти всегда является плохим стилем программирования.

Следующая немаловажная деталь - это процесс создания самой библиотеки. Статические библиотеки создаются при помощи архиватора **ar**, а совместно используемые - при помощи **gcc** с опцией **-shared**. В данном случае **gcc** опять же вызывает линковщик, но не для сборки бинарника, а для создания динамической библиотеки.

Последнее отличие - опции **-fPIC (-fpic)** при компиляции **h_world.c** и **g_world.c**. Эта опция сообщает компилятору, что объектные файлы, полученные в результате компиляции должны содержать позиционно-независимый код (PIC - Position Independent Code), который используется в динамических библиотеках. В таком

коде используются не фиксированные позиции (адреса), а плавающие, благодаря чему код из библиотеки имеет возможность подключаться к программе в момент запуска.

3. Основы низкоуровневого ввода-вывода

3.1. Обзор механизмов ввода-вывода в Linux

В языке С для осуществления файлового ввода-вывода используются механизмы стандартной библиотеки языка, объявленные в заголовочном файле **stdio.h**. Как вы вскоре узнаете консольный ввод-вывод - это не более чем частный случай файлового ввода-вывода. В С++ для ввода-вывода чаще всего используются потоковые типы данных. Однако все эти механизмы являются всего лишь надстройками над низкоуровневыми механизмами ввода-вывода ядра операционной системы.

Функции ввода и вывода стандартной библиотеки Си позволяют читать данные из файлов или получать их с устройств ввода (например, с клавиатуры) и записывать данные в файлы или выводить их на различные устройства (например, на принтер).

Функции ввода/вывода делятся на три класса:

1. Ввод/вывод верхнего уровня (с использованием понятия "поток"). Для этих функций характерно, что они обеспечивают буферизацию работы с файлами. Это означает, что при чтении или записи информации обмен данными осуществляется не между программой и указанным файлом, а между программой и промежуточным буфером, расположенным в оперативной памяти.

При записи в файл информация из буфера записывается при его заполнении или при закрытии файла. При чтении данных программой информация берется из буфера, а в буфер она считывается при открытии файла и впоследствии каждый раз при опустошении буфера. Буферизация ввода/вывода выполняется автоматически, она позволяет ускорить выполнение программы за счет уменьшения количества обращений к сравнительно медленно работающим внешним устройствам.

Для пользователя файл, открытый на верхнем уровне, представляется как последовательность считываемых или записываемых байтов. Чтобы отразить эту особенность

организации ввода/вывода, предложено понятие "поток" (англ. stream). Когда файл открывается, с ним связывается поток, выводимая информация записывается "в поток", а считываемая берется "из потока".

Когда поток открывается для ввода/вывода, он связывается со структурой типа **FILE**, определенной с помощью **typedef** в файле **stdio.h**. Эта структура содержит свою необходимую информацию о файле. При открытии файла с помощью стандартной функции **fopen** возвращается указатель на структуру типа **FILE**. Этот указатель, называемый указателем потока, используете для последующих операций с файлом. Его значение передается всем библиотечным функциям, используемым для ввода/вывода через этот поток.

Функции верхнего уровня одинаково работают в различных операционных средах, с их помощью можно писать переносимые программы.

2. Ввод/вывод для консольного терминала путем непосредственного обращения к нему. Функции для консоли и порта распространяют возможности функций ввода/вывода верхнего уровня на соответствующий класс устройств, добавляя новые возможности.

Они позволяют читать или записывать на консоль (терминал) или в порт ввода/вывода (например, порт принтера). Эти функции читают или записывают данные побайтно. Для ввода или вывода с консоли устанавливаются некоторые дополнительные режимы, например, ввод с эхо-печатью символов или без нее, установка окна вывода, цветов текста и фона. Функции для консоли и порта являются уникальными для компьютеров, совместимых с IBM D PC.

3. Ввод/вывод низкого уровня (с использованием понятия "дескриптор"). Функции низкого уровня не выполняют буферизацию и форматирование данных, они позволяют непосредственно пользоваться средствами ввода/вывода операционной системы.

При низкоуровневом открытии файла с помощью функции

open с ним связывается файловый дескриптор (англ. handle). Дескриптор является целым значением, характеризующими размещение информации об открытом файле во внутренних таблицах операционной системы. Дескриптор используется при последующих операциях с файлом.

Функции низкого уровня из стандартной библиотеки обычно применяются при разработке собственных подсистем ввода/вывода. Большинство функций этого уровня переносимы в рамках некоторых систем программирования на Си, в частности, относящихся к операционной системе Unix и совместимым с ней.

С точки зрения модели КИС (Клиент-Интерфейс-Сервер), сервером стандартных механизмов ввода вывода языка С (**printf, scant, FILE*, fprintf, fputc** и т. д.) является библиотека языка. А сервером низкоуровневого ввода-вывода в Linux является само ядро операционной системы.

Пользовательские программы взаимодействуют с ядром операционной системы посредством специальных механизмов, называемых системными вызовами (system calls, syscalls). Внешне системные вызовы реализованы в виде обычных функций языка С, однако каждый раз вызывая такую функцию, мы обращаемся непосредственно к ядру операционной системы. Список всех системных вызовов Linux можно найти в файле **/usr/include/asm/unistd.h**. Здесь мы рассмотрим основные системные вызовы, осуществляющие ввод-вывод: **open(), close(), read(), write(), lseek()** и некоторые другие.

3.2. Файловые дескрипторы

В языке С при осуществлении ввода-вывода мы используем указатель **FILE***. Даже функция **printf()** в итоге сводится к вызову **vfprintf(stdout,...)**, разновидности функции **fprintf()**; константа **stdout** имеет тип **struct _IO_FILE***, синонимом которого является тип **FILE***. Это к тому, что консольный ввод-вывод - это файловый ввод-вывод.

Стандартный поток ввода, стандартный поток вывода и поток ошибок (как в С, так и в С++) - это файлы. В Linux все, куда можно что-то записать или откуда можно что-то прочитать представлено (или может быть представлено) в виде файла. Экран, клавиатура, аппаратные и виртуальные устройства, каналы, сокеты - все это файлы. Это очень удобно, поскольку ко всему можно применять одни и те же механизмы ввода-вывода, с которыми мы и познакомимся в этой главе. Владение механизмами низкоуровневого ввода-вывода дает свободу перемещения данных в Linux. Работа с локальными файловыми системами, межсетевое взаимодействие, работа с аппаратными устройствами, - все это осуществляется в Linux посредством низкоуровневого ввода-вывода.

Вы знаете, что при запуске программы в системе создается новый процесс (здесь есть свои особенности, о которых пока говорить не будем). У каждого процесса (кроме init) есть свой родительский процесс (parent process или просто parent), для которого новоиспеченный процесс является дочерним (child process, child). Каждый процесс получает копию окружения (environment) родительского процесса. Оказывается, кроме окружения дочерний процесс получает в качестве багажа еще и копию таблицы файловых дескрипторов.

Файловый дескриптор (file descriptor) - это целое число (int), соответствующее открытому файлу. Дескриптор, соответствующий реально открытому файлу всегда больше или равен нулю. Копия таблицы дескрипторов (читай: таблицы открытых файлов внутри процесса) скрыта в ядре. Мы не можем получить прямой доступ к этой таблице, как при работе с окружением через environ. Можно, конечно, кое-что "вытянуть" через дерево **/proc**, но нам это не надо. Программист должен лишь понимать, что каждый процесс имеет свою копию таблицы дескрипторов. В пределах одного процесса все дескрипторы уникальны (даже если они соответствуют одному и тому же файлу или устройству). В разных процессах дескрипторы могут совпадать или не совпадать - это не имеет никакого значения, поскольку у

каждого процесса свой собственный набор открытых файлов.

Возникает вопрос: сколько файлов может открыть процесс? В каждой системе есть свой лимит, зависящий от конфигурации. Если вы используете bash или ksh (Kom Shell), ТО можете воспользоваться внутренней командой оболочки **ulimit**, чтобы узнать это значение.

$ ulimit -n
1024
$

Если вы работаете с оболочкой C-shell (csh, tcsh), то в вашем распоряжении команда **limit**:

$ limit descriptors
descriptors 1024
$

В командной оболочке, в которой вы работаете (bash, например), открыты три файла: стандартный ввод (дескриптор **0**), стандартный вывод (дескриптор **1**) и стандартный поток ошибок (дескриптор **2**). Когда под оболочкой запускается программа, в системе создается новый процесс, который является для этой оболочки дочерним процессом, следовательно, получает копию таблицы дескрипторов своего родителя (то есть все открытые файлы родительского процесса). Таким образом, программа может осуществлять консольный ввод-вывод через эти дескрипторы..

Таблица дескрипторов, помимо всего прочего, содержит информацию о текущей позиции чтения-записи для каждого дескриптора. При открытии файла, позиция чтения-записи устанавливается в ноль. Каждый прочитанный или записанный байт увеличивает на единицу указатель текущей позиции.

3.3. Открытие, файла: системный вызов open()

Чтобы получить возможность прочитать что-то из файла или записать что-то в файл, его нужно открыть. Это делает

системный вызов **open()**. Этот системный вызов не имеет постоянного списка аргументов (за счет использования механизма **va_arg**); в связи с этим существуют две "разновидности" **open()**. Не только в C++ есть перегрузка функций. Если интересно, то о механизме **va_arg** можно прочитать на man-странице stdarg (man 3 stdarg) или в книге Б. Кернигана и Д. Ритчи "Язык программирования Си". Ниже приведены адаптированные прототипы системного вызова **open()**.

int open (const char * filename, int flags, mode_t mode);
int open (const char * filename, int flags);

Системный вызов **open()** объявлен в заголовочном файле **fcntl.h**. Ниже приведен общий адаптированный прототип **open()**.

int open (const char * filename, int flags,...);

Начнем по порядку. Первый аргумент - имя файла в файловой системе в обычной форме: полный путь к файлу (если файл не находится в текущем каталоге) или сокращенное имя (если файл в текущем каталоге).

Второй аргумент - это режим открытия файла, представляющий собой один или несколько флагов открытия, объединенных оператором побитового ИЛИ. Список доступных флагов приведен в Таблице 4 Приложения 1. Наиболее часто используют только первые семь флагов. Если вы хотите, например, открыть файл в режиме чтения и записи, и при этом автоматически создать файл, если такового не существует, то второй аргумент open() будет выглядеть примерно так:

O_RDWR|O_CREAT

Константы-флаги открытия объявлены в заголовочном файле **bits/fcntl.h**, однако не стоит включать этот файл в свои программы, поскольку он уже включен в файл **fcntl.h**.

Третий аргумент используется в том случае, если **open()** создает новый файл. В этом случае файлу нужно задать права доступа (режим), с которыми он появится в файловой системе. Права доступа задаются перечислением флагов, объединенных побитовым ИЛИ. Вместо флагов можно использовать число (как правило восьмиричное), однако первый способ нагляднее и

предпочтительнее. Список флагов приведен в Таблице 1 Приложения 1. Чтобы, например, созданный файл был доступен в режиме "чтение-запись" пользователем и группой и "только чтение" остальными пользователями, - в третьем аргументе **open()** надо указать примерно следующее: **S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH** или **0664**. Флаги режима доступа реально объявлены в заголовочном файле **bits/stat.h**, но он не предназначен для включения в пользовательские программы, и вместо него мы должны включать файл **sys/stat.h**. Тип **mode_t** объявлен в заголовочном файле **sys/types.h**.

Если файл был успешно открыт, **open()** возвращает файловый дескриптор, по которому мы будем обращаться к файлу. Если произошла ошибка, то **open()** возвращает **-1.**

3.4. Закрытие файла: системный вызов close()

Системный вызов **close()** закрывает файл. Вообще говоря, по завершении процесса все открытые файлы (кроме файлов с дескрипторами **0**, **1** и **2**) автоматически закрываются. Тем не менее, это не освобождает нас от самостоятельного вызова **close()**, когда файл нужно закрыть. К тому же, если файлы не закрывать самостоятельно, то соответствующие дескрипторы не освобождаются, что может привести к превышению лимита открытых файлов. Простой пример: приложение может быть настроено так, чтобы каждую минуту открывать и перечитывать свой файл конфигурации для проверки обновлений. Если каждый раз файл не будет закрываться, то в моей системе, например, приложение может "накрыться медным тазом" примерно через 17 часов. Автоматически! Кроме того, файловая система Linux поддерживает механизм буферизации. Это означает, что данные, которые якобы записываются, реально записываются на носитель (синхронизируются) только через какое-то время, когда система сочтет это правильным и оптимальным. Это повышает

производительность системы и даже продлевает ресурс жестких дисков. Системный вызов **close()** не форсирует запись данных на диск, однако дает больше гарантий того, что данные останутся в целости и сохранности.

Системный вызов **close()** объявлен в файле **unistd.h**. Ниже приведен его адаптированный прототип:

int close (int fd);

Очевидно, что единственный аргумент - это файловый дескриптор. Возвращаемое значение **0** в случае успеха, и **1** - в случае ошибки. Довольно часто **close()** вызывают без проверки возвращаемого значения. Это не очень грубая ошибка, но, тем не менее, иногда закрытие файла бывает неудачным (в случае неправильного дескриптора, в случае прерывания функции по сигналу или в «случае ошибки ввода-вывода, например). В любом случае, если программа сообщит пользователю, что файл невозможно закрыть, это хорошо.

Теперь можно написать простенькую программу, использующую системные вызовы **open()** и **close()**. Мы еще не умеем читать из файлов и писать в файлы, поэтому напишем программу, которая создает файл с именем, переданным в качестве аргумента (**argv[1]**) и с правами доступа **0600** (чтение и запись для пользователя). Ниже приведен исходный код программы.

```
/* openclose.c */
#include <fcntl.h> /* open() and O_XXX flags */
#include <sys/stat.h> /* S_IXXX flags */
#include <sys/types.h> /* mode_t */
#include <unistd.h> /* close() */
#include <stdlib.h>
#include <stdio.h>
int main (int argc, char ** argv)
{
        int fd;
        mode_t mode = S_IRUSR | S_IWUSR;
        int flags = O_WRONLY | O_CREAT | O_EXCL;
        if (argc < 2)
```

```
{
        fprintf (stderr, "openclose: Too few arguments\n”);
        fprintf (stderr, "Usage: openclose <filename>\n");
        exit (1);
}
fd = open (argv[1], flags, mode);
if(fd<0)
{
        fprintf (stderr, "openclose: Cannot open file '%s'\n",
argv[1]);
        exit(1);
}
if (close (fd)!=0)
{
        fprintf (stderr, "Cannot close file (descriptor=%d)\n",
fd);
        exit(1);
}
exit (0);
}
```

Обратите внимание, если запустить программу дважды с одним и тем же аргументом, то на второй раз **open()** выдаст ошибку. В этом виноват флаг **O_EXCL** (см. Таблицу 4 Приложения 1), который "дает добро" только на создание еще не существующих файлов. Наглядности ради, флаги открытия и флаги режима мы занесли в отдельные переменные, однако можно было бы сделать так:

```
fd = open (argv[1], O_WRONLY | O_CREAT | O_EXCL, S_IRUSR |
S_IWUSR);
```

Или так:

```
fd = open (argv[1], O_WRONLY | O_CREAT |O_EXCL, 0600);
```


3.5. Чтение файла: системный вызов read()

Системный вызов **read()**, объявленный в файле **unistd.h**, позволяет читать данные из файла. В отличие от библиотечных функций файлового ввода-вывода, которые предоставляют возможность интерпретации считываемых данных. Можно, например, записать в файл следующее содержимое: 2006.

Теперь, используя библиотечные механизмы, можно читать файл по-разному:

fscanf(filep, "%s", buffer);

fscanf (filep, "%d", number);

Системный вызов **read()** читает данные в "сыром" виде, то есть как последовательность байт, без какой-либо интерпретации. Ниже представлен адаптированный прототип **read()**.

ssize_t read (int fd, void * buffer, size_t count);

Первый аргумент - это файловый дескриптор. Здесь больше сказать нечего. Второй аргумент - это указатель на область памяти, куда будут помещаться данные. Третий аргумент - количество байт, которые функция **read()** будет пытаться прочитать из файла. Возвращаемое значение -количество прочитанных байт, если чтение состоялось и **-1**, если произошла ошибка. Хочу заметить, что если **read()** возвращает значение меньше **count**, то это не символизирует об ошибке.

Несколько слов о типах. Тип **size_t** в Linux используется для хранения размеров блоков памяти. Какой тип реально скрывается за **size_t**, зависит от архитектуры; как правило это **unsigned long int** или **unsigned int**. Тип **ssize_t** (Signed SIZE Type) - это тот же **size_t**, только знаковый. Используется, например, в тех случаях, когда нужно сообщить об ошибке, вернув отрицательный размер блока памяти. Системный вызов **read()** именно так и поступает.

Теперь напишем программу, которая просто читает файл и выводит его содержимое на экран. Имя файла будет передаваться в качестве аргумента (**argv[1]**). Ниже приведен исходный код этой

программы.

```
/* myread.c */
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
int main (int argc, char ** argv)
{
int fd;
ssize_t ret;
char ch;
fd = open (argv[1], O_RDONLY);
if(fd<0)
{
fprintf (stderr, "Cannot open file\n");
exit(1);
}
while ((ret= read (fd, &ch, 1)) > 0)
{
putchar (ch);
}
if(ret<0)
{
        fprintf (stderr, "myread: Cannot read file\n");
        exit(1);
}
close (fd);
exit (0);
}
```

В этом примере используется укороченная версия **open()**, так как файл открывается только для чтения. В качестве буфера (второй аргумент **read()**) мы передаем адрес переменной типа char. По этому адресу будут считываться данные из файла (по одному

байту за раз) и передаваться на стандартный вывод. Цикл чтения файла заканчивается, когда **read()** возвращает нуль (нечего больше читать) или **-1** (ошибка). Системный вызов **close()** закрывает файл.

Как можно заметить, в нашем примере системный вызов **read()** вызывается ровно столько раз, сколько байт содержится в файле. Иногда это действительно нужно; но не здесь. Чтение-запись посимвольным методом (как в нашем примере) значительно замедляет процесс ввода-вывода за счет многократных обращений к системным вызовам. По этой же причине возрастает вероятность возникновения ошибки. Если нет действительной необходимости, файлы нужно читать блоками. Ниже приведен исходный код программы, которая делает то же самое, что и предыдущий пример, но с использованием блочного чтения файла. Размер блока установлен в 64 байта.

```
/* myread1.c */
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#define BUFFER_SIZE 64
int main (int argc, char ** argv)
{
int fd;
ssize_t read_bytes;
char buffer[BUFFER_SIZE+1];
fd = open (argv[1], O_RDONLY);
if(fd<0)
fprintf (stderr, "Cannot open file\n");
exit(1);
read_bytes = read (fd, buffer, BUFFER_SIZE)) > 0)
……….

fprintf (stderr, "myread: Cannot read file\n");
```

```
exit(1);
}
close (fd);
exit (0);
}
```

Примечание: дополните самостоятельно недостающие строчки в программе **myread1.c**.

Теперь можно примерно оценить и сравнить скорость работы двух примеров. Для этого надо выбрать в системе достаточно большой файл (бинарник ядра или видеофильм, например) и посмотреть на то, как быстро читаются эти файлы:

```
$ time ./myread /boot/vmlinuz > /dev/null
real 0m 1.443s
user 0m0.383s
sys 0ml.039s
$ time ./myread 1 /boot/vmiinuz > /dev/null
real 0m0.055s
user 0m0.010s
sys 0m0.023s
$
```

Примечание: чтобы эти программы знали, какой именно файл следует открывать и читать, необходимо написать дополнительную (главную) программу, которая будет запрашивать имя этого файла и вызывать программу **myread** или **myread1**. Или же можно вместо **argv[1]** указать имя файла, заключенное в кавычки.

3.6. Запись в файл: системный вызов write()

Для записи данных в файл используется системный вызов write(). Ниже представлен его прототип.

ssize_t write (int fd, const void * buffer, size_t count);

Как видите, прототип **write()** отличается от **read()** только спецификатором **const** во втором аргументе. В принципе **write()** выполняет процедуру, обратную **read()**: записывает count байтов из буфера **buffer** в файл с дескриптором **fd**, возвращая количество записанных байтов или -1 в случае ошибки. Так просто, что можно сразу переходить к примеру. За основу возьмем программу **myread1** из предыдущего раздела.

```
/* rw.c */
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h> /* read(), write(), close() */
#include <fcntl.h> /* open(), O_RDONLY */
#include <sys/stat.h> /* S_IRUSR */
#include <sys/types.h> /* mode_t.*/
#define BUFFER_SIZE 64
int main (int argc, char ** argv)
{
int fd;
ssize_t read_bytes;
ssize_t written_bytes;
char buffer[BUFFER_SIZE];
fd = open (argv[1], 0_RDONLY);
if (fd<0)
fprintf (stderr, "Cannot open file\n");
exit(1);
while ((read_bytes = read (fd, buffer, BUFFER_SIZE)) > 0)
{
/* 1 == stdout */
written_bytes = write (1, buffer, read_bytes);
if(written_bytes!=read_bytes) ,
{
fprintf (stderr, "Cannot write\n");
exit(1);
}
}
if (read_bytes < 0)
{
```

```
fprintf (stderr, "myread: Cannot read file\n");
exit(1);
}
close (fd);
exit (0);
}
```

В этом примере нам уже не надо изощряться в попытках вставить нуль-терминатор в строку для записи, поскольку системный вызов **write()** не запишет большее количество байт, чем мы ему указали. В данном случае для демонстрации **write()** мы просто записывали данные в файл с дескриптором **1**, то есть в стандартный вывод. Но прежде, чем переходить к чтению следующего раздела, попробуйте самостоятельно записать что-нибудь (при помощи **write()**, естественно) в обычный файл. Когда будете открывать файл для записи, обратите пожалуйста внимание на флаги **O_TRUNC, O_CREAT** и **O_APPEND**. Подумайте, все ли флаги сочетаются между собой по смыслу.

3.7. Произвольный доступ: системный вызов lseek()

Как уже говорилось, с каждым открытым файлом связано число, указывающее на текущую позицию чтения-записи. При открытии файла позиция равна нулю. Каждый вызов **read()** или **write()** увеличивает текущую позицию на значение, равное числу прочитанных или записанных байт. Благодаря этому механизму, каждый повторный вызов **read()** читает следующие данные, и каждый повторный **write()** записывает данные в продолжение предыдущих, а не затирает старые. Такой механизм последовательного доступа очень удобен, однако иногда требуется получить произвольный доступ к содержимому файла, чтобы, например, прочитать или записать файл заново.

Для изменения текущей позиции чтения-записи используется системный вызов l**seek()**. Ниже представлен его прототип.

off_t lseek (int fd, ott_t offset, int against);

Первый аргумент, как всегда, - файловый дескриптор. Второй аргумент - смещение, как положительное (вперед), так и отрицательное (назад). Третий аргумент обычно передается в виде одной из трех констант **SEEK_SET**, **SEEK_CUR** и **SEEK_END**, которые показывают, от какого места отсчитывается смещение. **SEEK_SET** - означает начало файла, **SEEK_CUR** - текущая позиция, **SEEK_END** - конец файла. Рассмотрим следующие вызовы:

***lseek* (fd, 0, SEEK_SET);**
***lseek* (fd, 20, SEEK_CUR);**
***lseek* (fd, -10, SEEK_END);**

Первый вызов устанавливает текущую позицию в начало файла. Второй вызов смещает позицию вперед на 20 байт. В третьем случае текущая позиция перемещается на 10 байт назад относительно конца файла.

В случае удачного завершения, **lseek()** возвращает значение установленной "новой" позиции относительно начала файла. В случае ошибки возвращается **-1.**

Рассмотрим программу рисования символами. Программа оказалась не слишком простой, однако если вы сможете разобраться в ней, то можете считать, что успешно овладели азами низкоуровневого ввода-вывода Linux. Ниже представлен исходный код этой программы.

```
/* draw.c */
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h> /* memset() */
#define N_ROWS 15 /* Image height */
#define N_COLS 40 /* Image width */
#define FG_CHAR '0' /* Foreground character */
```

```
#define IMG_FN "image" /* Image filename */
#define N_MIN(A,B) ((A)<(B)?(A):(B))
#defme N_MAX(A,B) ((A)>(B)?(A):(B))
static char buffer[N_COLS];

void init_draw (int fd)
{
ssize_t bytes_written = 0;
memset (buffer,'', N_COLS);
buffer [N_COLS] = '\n';
while (bytes_written < (N_ROWS * (N_COLS+1)))
bytes_written += write (fd, buffer, N_COLS+1);
}

void draw_point (int fd, int x, int y)
{
char ch = FG_CHAR;
lseek (fd, y * (N_COLS+1) + x, SEEK_SET);
write (fd, &ch, 1);
}

void draw_hline (int fd, int y, int x1, int x2)
{
size_t bytes_write = abs (x2-x1) + 1;
memset (buffer, FG_CHAR, bytes_write);
lseek (fd, y * (N_COLS+1) + N_MIN (x1, x2), SEEK_SET);
write (fd, buffer, bytes_write);
}

void draw_vline (int fd, int x, int y1, int y2)
{
inti=N_MIN(yl,y2);
while (i <= N_MAX(y2, y1)) draw_point (fd, x, i++);

int main (void)
```



```
{
int a, b, c, i = 0;
char ch;
int fd = open (IMG_FN, O_WRONLY | O_CREAT | O_TRUNC,
0644);
if(fd<0)
{
fprintf (stderr, "Cannot open file\n");
exit(1);
}
init_draw (fd);
char * icode[] = { "v 1 1 11", "v 11 7 11", "v 14 5 11",
"v 18 6 11", "v 21 5 10", "v 25 5 10", "v 29 5 6", "v 33 5 6",
"v 29 10 11", "v 33 10 11", "h 11 1 8","h 5 16 17",
"h 11 22 24", "p 11 5 0", "p 15 6 0", "p 26 11 0", "p 30 7 0",
"p 32 7 0", "p 31 8 0", "p 30 9 0", "p 32 9 0", NULL };
while (icode[i]!= NULL)
{
sscanf(icode[i], "%c %d %d %d", &ch, &a, &b, &c);
switch (ch) {
case 'v': draw_vline (fd, a, b, c); break;
case 'h': draw_hline (fd, a, b, c); break;
case 'p': draw_point (fd, a, b); break;
default: abort();
}
i++;
} close (fd);
exit (0);
```

Теперь разберемся, как работает эта программа. Изначально "полотно" заполняется пробелами. Функция **init_draw()** построчно записывает в файл пробелы, чтобы получился "холст размером **N_ROWS** на **N_COLS**. Массив строк **icode** в функции **main()** - это набор команд рисования. Команда начинается с одной из трех литер: 'v' - нарисовать вертикальную линию, 'h' -нарисовать горизонтальную линию, 'p' - нарисовать точку. После каждой такой

литеры следуют три числа. В случае вертикальной линии первое число - фиксированная координата X, а два других числа - это начальная и конечная координаты Y. В случае горизонтальной линии фиксируется координата Y (первое число). Два остальных числа - начальная координата X и конечная координата X. При рисовании точки используются только два первых числа: координата X и координата Y. Итак, функция **draw_vline()** рисует вертикальную линию, функция **draw_hline()** рисует горизонтальную линию, а **draw_point()** рисует точку.

Функция **init_draw()** пишет в файл **N_ROWS** строк, каждая из которых содержит **N_COLS** пробелов, заканчивающихся переводом строки. Это процедура подготовки "холста".

Функция **draw_point()** вычисляет позицию (исходя из значений координат), перемещает туда текущую позицию ввода-вывода файла, и записывает в эту позицию символ (**FG_CHAR**), которым мы рисуем "картину".

Функция **draw_hline()** заполняет часть строки символами **FG_CHAR**. Так получается горизонтальная линия. Функция **draw_vline()** работает иначе. Чтобы записать вертикальную линию, нужно записывать по одному символу и каждый раз "перескакивать" на следующую строку. Эта функция работает медленнее, чем **draw_hline()**, но иначе мы не можем.

Полученное изображение записывается в файл **image**. Будьте внимательны: разгрузить исходный код, из программы исключены многие проверки (**read()**, **write()**, диапазон координат и проч.). Попробуйте включить эти проверки самостоятельно.

4. Разработка системной оболочки

Модуль 1. Основной модуль

Вначале перечисляем библиотеки, которые будут необходимы для работы основного модуля. Основным он называется потому, что он начинает работать в самом начале после запуска программы – системной оболочки. Из него уже вызываются процедуры, работающие с файлами и каталогами, такие, как: удаление файла, копирование файла, удаление каталога и т.д.

```
#include <cstdlib>
#include <iostream>
#include <string.h>
```

Для успешной сдачи задания следует разобраться, что представляет собой и для чего предназначена КАЖДАЯ из приведенных как выше, так и ниже, библиотек.

Далее следует подключить процедуры, которые будут выполнять те или иные действия с файлами или каталогами. Отметим, что несмотря на то, что в Linux идеология работы с файлом и каталогом одинакова, одни и те же типы процедур для файла и каталога должны быть написаны по отдельности. Дело, по крайней мере, в том, что функции будут разными. Например, для удаления файла можно использовать процедуру удаления жестких ссылок **unlink**, тогда как для удаления каталога – процедуру **rmdir.** Другое дело, что обе функции можно объединить в одном программном модуле; если файл – то вызывается **unlink,** если каталог, то – **rmdir**. Правда, предварительно необходимо будет обозначить процедуру проверки свойств объекта: файл это или каталог и, в соответствии с результатом проверки, вызывать требующуюся процедуру.

```
#include "err1.h" /*Модуль – обработчик ошибок*/
#include "df.h" /удаляет файл/
#include "md.h" /создает каталог/
```

```
#include "cp.h" /копирует файл/
#include "of.h" /открывает файл/
#include "rd.h" /переименовывает  каталог/
#include "rf.h" /переименовывает  файл/
#include "stfile.h" /выдает информацию о файле/
using namespace std;
```

Теперь можно начать писать основную часть программы – главного модуля.

```
int main()
char path[300];
char nname[300];
char c[50];
```

После описания требующихся массивов создаем блок, который будет выбирать одну из процедур для обработки файлов или каталогов:

```
cout<<"if you want delete file type \"del-f\"\n";
cout <<"if you want open file type \"op-f\"\n";
cout <<"if you want withdraw file stats type \"st-f\"\n";
cout <<"if you want copy file type \"cp-f\"\n";
cout <<"if you want rename file type \"rn-f\"\n";
cout <<"if you want delete dir type \"del-d\"\n";
cout <<"if you want create dir type \"cr-d\"\n";
cout <<"if you want exit type \"exit\"\n";
```

Это – часть, «задающая вопросы» о том, что мы хотим сделать. Теперь рабочая часть (построенная в виде бесконечного цикла). В зависимости от того, что именно мы введем из предложенного выше списка, на выполнение будет вызываться та или иная функция (описанная ниже в рамках цикла **while**).

```
while(TRUE)
{
        cin>>c;
```

В зависимости от того, чему равна переменная **c**, в рамках этого цикла вызывается та или иная процедура. Например, если эта переменная равна **"exit"**, то производится выход из цикла и программа прекращает работу. Если **c= del-f**, вызывается функция

delfile удаления файла, и т.д.

```
        if(strcmp(c, "exit")==0)
        return 0;
        if(strcmp(c, "del-f ")==0)
        {
        cout<<"Enter path name\n";
        cin>>path;
        delfile(path);
        cout<<"Enter command\n";
        }
```

Здесь вызывается разрабатываемая нами функция **delfile**, см ниже. Параметром этой функции является переменная **path**. Тем самым, здесь, в основном модуле мы лишь задаем значение переменной **path**, передаем в функцию **delfile**; в основном модуле НЕ происходит непосредственно удаления файла. А вот при вызове функции **delfile** происходит ее работа, которая как раз и заключается в удалении файла.

После чего программа пишет **Enter command** (введите команду). Т.е. после того, как процедура удаления файла закончила свою работу, предлагается ввести новую команду (например, rn-f; cr-d и т.д.); после ввода команды мы возвращаемся в начало цикла **while,** затем вновь выполняем команду в зависимости от того, какую именно команду выбрали. И т.д. Так происходит до тех пор, пока в качестве переменной **c** не будет выбрано **exit**. После чего программа завершит свою работу.

Рассмотрим далее основной модуль.

```
if(strcmp(c, "op-f")==0)
{
        cout<<"Enter path name\n";
        cin>>path;
        openfile(path);
        cout<<" Enter command\n";
}
        if(strcmp(c,"st-f")==0)
{
```

```
        cout<<" Enter path name\n";
        cin>>path;
        statfile(path);
        cout<<"Enter command\n";
}
        if(strcmp(c,"cp-f")==0)
{
        cout<<" Enter path name\n";
        cin>>path;
        cout<<"enter new name\n";
        cin>>nname;
        copyfile(path, nname);
        cout<<"Enter command\n";
}
```

В вышеприведенной процедуре вводились две переменные, так как эта процедура вызывает копирование файла; поэтому для нее необходимо имя старого файла (который хотим скопировать) и имя нового файла (в который хотим скопировать). Как видно, процедура **copyfile** имеет два параметра.

```
if(strcmp(c, "rn-f")==0)
{
        cout<<"Enter path name\n";
        cin>>path;
        cout<<"enter new name\n";
        cin>>nname;
        renamefile(path,nname);
        cout<<"Enter command\n";
}
        if(strcmp(c, "del-d")==0)
{
        cout<<" Enter path name\n";
        cin>>path;
        deldir(path);
        cout<<"Enter command\n";
}
```



```
        if(strcmp(c,"cr-d")==0)
        {
        cout<<" Enter path name\n";
        cin>>path;
        makedir(path);
        cout<<"Enter command\n";
        }
}
        return 0;
        }
```

В данном модуле не используются никакие специфические функции или процедуры. Только стандартные и написанные нами. Задача основного модуля - используя остальные модули выполнять то, что требуется от системной оболочки. Основной модуль передаёт некоторое значение (в данном случае он передаёт пути к файлам или каталогам) в написанные нами процедуры (которые находятся в модулях) в зависимости от параметров входной строки. То есть наш основной модуль выполняет функции командного интерпретатора.

В нем содержатся вызовы функций, работающих с файлами и каталогами, например, **err1, df, md** и др. На данный момент эти функции еще отсутствуют, их следует создать. Пока же, чтобы проверить работу основного модуля, целесообразно создать лишь (пустые) формы этих функций, т.е. заготовки. Вызов их не будет производить никаких действий. Но в дальнейшем, по мере реализации всех функций, они будут работать.

Например, для функции (модуля) открытия файла можно предварительно написать следующую заготовку:

```
void openfile(char *path)
{
        continue;
}
```

и т.д., для всех функций, для которых в основном модуле имеются вызовы.

См. также Приложение 2 (необходимо создать

соответствующие заголовочные файлы – для каждой функции).

Впоследствии, по мере реализации каждой из функций, их работу можно будет проверять, запуская основной модуль и вызывая соответствующие операции. Например, для вызова функции открытия файла из основного модуля после его запуска следует ввести **op-f** и нажать **Enter**.

Примечание: не забывайте заново осуществлять компиляцию и сборку основного модуля всякий раз, когда Вы изменили хотя бы одну из функций, к которым он обращается! С целью экономии времени, это целесообразно автоматизировать на основе использования **Makefile**. Как это делается, описано в первой части настоящих методических указаний.

Модуль 2. Вывод описания ошибки

Рассмотрим код функции, позволяющей узнать описание последней ошибки.

```
#include <iostream>
#include <errno.h>
#include <string.h>
#include "err1.h"
using namespace std;
void err1()
{
        cout<<"Error: "<<strerror(errno)<<"\n";   return;
}
```

Структура модуля вполне классическая, сначала идёт подключение необходимых для работы функций библиотек, потом уже описание самой функции.

В данном модуле нет ничего особенного, но стоит отдельно рассмотреть

Строку **cout<<"Error:"<<strerror(errno)<<"\n";**

Она выводит сообщение с ошибкой, которое получает от функции **strerror**. Данная функция по номеру ошибки возвращает

строковое описание ошибки. А номер ошибки она получает из функции **errno**, которая возвращает номер последней ошибки.

Эта функция нам понадобится дальше, при написании процедур, выполняющих действия над файлами или каталогами. Если в процессе их выполнения будут возникать ошибки, мы, благодаря этой функции, будем знать, какая именно ошибка произошла.

Модуль 3. Открытие файла

```
#include <cstdlib>
#include <iostream>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "err1.h"
#include "of.h"
using namespace std;

void openfile(char *path)
{
        int fd;
        if(fd=open(path,O_CREAT, S_IRWXU)==-1)
        err1();
else cout<<"File opening/creation succesfull\n";
}
```

Обратим внимание, что оператор **open** в качестве параметров использует переменную **path** (ту самую, которую мы задавали в основном модуле), а также так называемые флаги (**O_CREAT, S_IRWXU**). Последние определяют права, при которых открыт файл, имя которого задано переменной **path**. Подробнее про флаги, а также права, с которыми может быть открыт файл, следует обязательно прочитать где-нибудь. Например, в сети интернет или книге под названием типа

"Программирование на Си для Linux».

Стоит также сказать пару слов про подключаемые библиотеки. Нас интересуют

#include "err1.h"

#include "of.h"

Первая строчка подключает заголовочный файл процедуры вывода ошибки, чтобы мы могли использовать её. Вторая же содержит прототип процедуры, которую мы описываем. Подробная информация про заголовочные файлы находится в Приложении 2.

Рассмотрим теперь саму процедуру открытия файла. Наибольший интерес вызывает строчка

if(fd=open(path, O_CREAT, S_IRWXU)==-1)

Эта строка и выполняет основную работу процедуры. В ней одновременно происходит открытие (создание, если такого файла нет) файла, присваивание переменной значения файлового дескриптора **fd**, проверка на ошибки. Рассмотрим теперь каждый этап подробнее.

С помощью функции **open** происходит попытка открытия файла. В данном случае используется связка флагов **O_CREAT, S_IRWXU**, которая указывает функции, что, в случае, если файл не обнаружен, то необходимо его создать (первый флаг). Далее этому файлу присваиваются права доступа: для записи, для чтения. Далее функция **open** передаёт значение в переменную **fd**, так же происходит проверка на ошибку (если функция вернула значение -1, значит произошла ошибка). При этом вызывается разработанная нами процедура **err1.**

Модуль 4. Удаление файла

```
#include <cstdlib>
#include <unistd.h>
#inciude <iostream>
#include <sys/types.h>
#include <sys/stat.h>
```



```
#include <fcntl.h>
#include "df.h"
#include "err1.h"
using namespace std;
void delfile(char *path)
{
        if(unlink(path)==-1)
        err1();
        else cout<<"File deleting succesfull";}
```

Данная процедура должна удалять файл. Делает она это, большей частью, с помощью функции **unlink.**

unlink удаляет имя из файловой системы. Если это имя было последней ссылкой на файл и больше нет процессов, которые держат этот файл открытым, данный файл удаляется и место, которое он занимает освобождается для дальнейшего использования.

Если имя было последней ссылкой на файл, но какие-либо процессы всё ещё держат этот файл открытым, файл будет оставлен, пока последний файловый дескриптор, указывающий на него, не будет закрыт.

Если функция завершилась успешно, то она возвращает 0, в противном случае -1.

Модуль 5. Переименование файла

```
#include <cstdlib>
#include <iostream>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include "rf.h"
#include "errl.h"
```

```
using namespace std;

#define BS 1
char buf1[BS];

void renamefile(char *path, char *nname)
{
        int i=0, ffd, sfd;
        ssize_t count;
        ffd=open(path, O_RDONLY);
        if(ffd==-1)
        err1();
        sfd=open(nname,      O_WRONLY|O_CREAT|O_EXCL,
0644);
```

Обратите внимание на цифры 0644, означающие права на файл, под которыми он открывается (или создается). Следует обязательно выяснить, что это за цифры, какие они бывают вообще, для чего они. Почему эти цифры не используются при открытии исходного файла?

```
        if(sfd==-1)
        err1();
        while(count=read(ffd,buf1,BS)>0)
```

Из первого файла (т.е. по файловому дескриптору **ffd**) считывается в буфер (который представляет собой массив) **buf1** число байт, равное **BS.**

```
        {
                write(sfd, buf1, count);
                i++;
        }
        if(count==-1)
        err1();
        cout<<"Copy succesfull\n";
        cout<<"Bytes copyed:"<<i;
        close(ffd);
```



```
        unlink(path);
        cout<<"\nSource file sucsefully deleted\n";
        close(sfd);
}
```

Данная процедура работает по следующему алгоритму

1. Открытие файла для чтения
2. Создание нового файла
3. Копирование информации из первого файла (исходного) во второй(новый)
4. Удаление первого файла

Пункты 1,2,3 уже были рассмотрены, как отдельные модули. Рассмотрим теперь пункт 4

```
while(count=read(ffd, buf1, BS)>0)
{
        write(sfd, buf1, count);
        i++;
}
```

Для понимания работы данного участка кода необходимо знать функции **read** и **write.**

read() пытается записать count байтов файлового описателя **fd** в буфер, адрес которого начинается с **buf.**

Если количество **count** равно нулю, то **read()** возвращает это нулевое значение и завершает свою работу. Если **count** больше, чем **SSIZE_MAX**, то результат не может быть определен. Иными словами, чтение из первого файла и запись во второй продолжается до тех пор, пока переменная **count** не станет равной 0.

При успешном завершении вызова возвращается количество байтов, которые были считаны (нулевое значение означает конец файла), а позиция файла увеличивается на это значение. Если количество прочитанных байтов меньше, чем количество запрошенных, то это не считается ошибкой:

например, данные могли быть почти в конце файла, в канале, на терминале, или **read()** был прерван сигналом. В случае ошибки возвращаемое значение равно -1, а переменной **errno** присваивается номер ошибки. В этом случае позиция файла не

определена.

write записывает до **count** байтов из буфера **buf1** в файл, на который ссылается файловый описатель **fd.** POSIX указывает на то, что вызов **write(),** произошедший после вызова **read()** возвращает уже новое значение. Заметьте, что не все файловые системы соответствуют стандарту POSIX.

В случае успешного завершения возвращается количество байтов, которые были записаны (ноль означает, что не было записано ни одного байта). В случае ошибки возвращается -1, а переменной **errno** присваивается соответствующее значение. Если **count** равен нулю, а файловый описатель ссылается на обычный файл, то будет возвращен ноль и больше не будет произведено никаких действий. Для специальных файлов результаты не могут быть перенесены на другую платформу.

Теперь понятно, что тот участок кода просто в цикле побайтово копирует информацию из одного файла в другой.

Далее, по алгоритму, следует удаление исходного файла, что в итоге даст нам, по сути, тот же файл, но с другим именем.

Модуль 6. Копирование файла

Процедура копирования файла ничем принципиально не отличается от -процедуры переименования файла. Единственное отличие- в процедуре копирования файла не происходит удаление исходного файла.

```
#include <cstdlib>
#include <iostream>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include "errl.h"
#include "cp.h"
using namespace std;
```



```
#define BS 1
char buf[BS];
void copyfile(char *path, char *nname)
{
        int i=0, ffd, sfd;
        ssize__t count;
        ffd=open(path,O_RDONLY);
        if(ffd==-1)
        err1();
        sfd=open(nname,O_WRONLY|O_CREAT|O_EXCL,064
4);
        if(sfd==-1)
        err1();
        while(count=read(ffd,buf,BS)>0)
        {
                write(sfd,buf,count);
                i++;
        }
        if(count==-1) err1();
        cout<<"Copy succefull\n";
        cout<<" Bytes copyed:\n"<<i;
        close(ffd);
        close(sfd);
}
```

Как видно, эта программа весьма незначительно отличается от предыдущей программы.

Модуль 7. Создание каталога

```
#include <cstdlib>
#include <iostream>
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include "err1.h"
using namespace std;

void makedir(char *path)
{
        if(mkdir(path,0777)==0)
                cout<<" Directory created\n";
        else err1();
}
```

В данном модуле нет ничего сложного, создание каталога происходит с помощью функции **mkdir**

*int mkdir (const char *pathname, mode_t mode);*

mkdir пытается создать каталог, который называется **pathname**.

mode задает права доступа, которые получит свежесозданный каталог. Эти права стандартным образом модифицируются с помощью **umask**; права доступа оказываются равны (**mode & ~umask**).

Свежесозданный каталог принадлежит фактическому владельцу процесса. Если на родительском каталоге установлен флаг **setgid**, или файловая система смонтирована с семантикой групп в стиле BSD, то новый каталог унаследует группу-владельца от своего родительского каталога; в противном случае группой-владельцем станет фактическая группа процесса.

Если у родительского каталога установлен бит **setgid**, то он будет установлен также и у свежесозданного каталога.

Конечно, ещё стоит прояснить момент с правами доступа, в нашем случае у каталога будут права 0777. Подробнее про маску доступа можно прочитать в Приложении 1, 3.

Модуль 8. Удаление каталога

Удаление каталога может показаться простой задачей. Но, в случае, если необходимо удалить вложенные каталоги и файлы, эта

задача слегка усложняется. Рассмотрим модуль.

```
#include <cstdlib>
#include <unistd.h>
#include <iostream>
#include <sys/types.h>
#include <dirent.h>
#include <errno.h>
#include <string.h>
#include "errl.h"
#inctude "rd.h"
using namespace std;

void deldir(char *path)
{
        DIR *dir;
        int u;
        struct dirent *sdir;
        dir=opendir(path);
        if(dir==NULL) err1();
        chdir(path);
        while((sdir=readdir(dir))!=NULL)
    {
        //cout << sdir->d_name<<"\n";
        u=unlink(sdir->d_name);
        if(u==-1 && errno==EISDIR)
    {
        if(strcmp(".", sdir->d_name)==0 || strcmp("..", sdir-
        >d_name)==0)
                continue;
        deldir(sdir->d_ name);
    }
        if(u==-1 && errno==!EISDIR)
                err1();
    }
        closedir(dir);
```

```
        chdir("..");
        if(rmdir(path)==-1) err1();
}
```

Данную процедуру нелегко понять, просто взглянув на неё, так что рассмотрим все строки с неизвестными операторами.

```
DIR *dir; int u;
struct dirent *sdir;
dir=opendir(path);
if(dir==NULL) err1();
chdir(path);
```

На данном участке кода происходит некоторого рода инициализация каталога. Он открывается, проверяется успешность попытки его открытия, текущий каталог меняется на данный.

```
while((sdir=readdir(dir))!=NULL)
{
        u=unlink(sdir->d_name);
        if(u==-1 && errno==EISDIR)
  {
        if(strcmp(".",sdir->d__name)==0 || strcmp("..", sdir->d_name)==0)
                continue;
        deldir(sdir->d_ name);
  }
        if(u==-l && errno==!EISDIR)
                err1();
}
```

Этот же участок без сомнения можно назвать самым важным в данном модуле. Здесь происходит рекурсивное удаление каталога. Рассмотрим его работу по подробнее.

```
while((sdir=readdir(dir))!=NULL)
```

Наш основной цикл. Те операторы, которые находятся в нём, будут выполняться, пока не будет достигнут конец каталога.

```
u=unlink(sdir->d_name);
```

Попытка удаления (к слову говоря, употреблять здесь слово «удаление» в его общем смысле не совсем верно, функция unlink

удаляет ссылки на файл, а уже потом освобождается дисковое пространство) файла. По результатам этой попытки наш модуль будет вести так или иначе.

```
if(u==-1 && errno==EISDIR)
{
if(strcmp(".", sdir->d_name)==0 || strcmp("..", sdir->d_name)==0)
        continue;
deldir(sdir->d_name);
}
if(u==-1 && errno==!EISDIR)
        err1();
```

Если наша попытка не удалась (то есть функция **unlink** вернула значение -1) и при этом последняя ошибка «**EISDIR**» (то есть ошибка «Это каталог»), то проверяем не является ли имя «.» или «..» (про эти имена можно более подробно прочитать в справочной литературе, скажу лишь что они обозначают родительский и текущий каталоги).Если имена совпали, то мы переходим на следующий итерационный шаг. Иначе просто заново входим в наш модуль (здесь и появляется рекурсия) передавая имя текущего каталога.

Если же попытка удаления не успешна и при этом ошибка не «**EISDIR**» то выводим ошибку.

```
closedir(dir);
chdir("..");
if(rmdir(path)==-1)
        err1();
}
```

Здесь же происходит соответственно закрытие каталога, переход в родительский каталог, удаление каталога (удаление в самом простом смысле, когда он пустой).

Таким образом, рекурсивный вызов заключается в том, что функция **deldir** вызывает саму себя до тех пор, пока программа не откроет такой каталог, который не содержит иных каталогов. При этом программа осуществляет удаление всех файлов в этом

каталоге, выходит из этого (теперь пустого) каталога и удаляет его. И, так далее, до тех пор, пока не выйдет из исходного каталога и не удалит его.

Модуль 9. Информация о файле или каталоге

Не смотря на довольно таки массивный код, процедура проста.

```
#include <cstdlib>
#include <iostream>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include "stfile.h"
#include "err1.h"
using namespace std;

void statfile(char *path)
{
struct stat sb;
if(stat(path,&sb)==-1)
        err1();
else
{
printf("File type:          "):
switch (sb.st_mode & S_IFMT)
{
        case S_IFBLK: cout<<" block device\n"; break;
        case S_IFCHR: cout<<"character device\n"; break;
        case S_IFDIR: cout<<"directory\n"; break;
        case S_IFIFO: cout<<"FIFO/pipe \ n "; break;
        case S_IFLNK: cout<<"symlink\n"; break;
        case S_IFREG: cout<<"regularfile\n"; break;
```



```
        case S_IFSOCK: cout<<"socket\n"; break;
        default: cout<<"unknown?\n"; break;
}
printf("i-node number: %ld\n", (long) sb.st_ino);

printf("Mode:      %lo(octal)\n",
        (unsigned long) sb.st_mode);
if(sb.st_mode & S_IRUSR) cout<<" owner has read permission\n";
if(sb.st_mode & S_IWUSR) cout<<" owner has write permission\n";
if(sb.st_mode & S_IXUSR & !S_IFDIR) cout<<"owner has execute permission\n";
if(sb.st__mode & S_IRGRP) cout<<"group has read permission\n";
if(sb.st_mode & S_IWGRP) cout<<"group has write permission\n";
if(sb.st_mode & S_IXGRP & !S_IFDIR) cout<<"group has execute permission\n";
if(sb.st_mode & S_IROTH) cout<<"others have read permission\n";
if(sb.st_mode & S_IWOTH) cout<<"others have write permission\n";
if(sb.st_mode & S_IXOTH & !S_IFDIR) cout<<"others have execute permission\n";
```

Далее следует правильно вывести на экран результаты анализа свойств файла (каталога).

```
printf("Link count: %ld\n", (long) sb.st_nlink);
printf("Ownership: UID=%ld GID=%ld\n",
        (long) sb.st_uid, (long) sb.st_gid);
printf("Preferred I/O block size: %ld bytes\n",
(long) sb.st_blksize);1
printf(" File size: %lld bytes\n",
(long long) sb.st_size);
printf(" Blocks allocated: %lld\n",
```

(long long) sb.st_blocks);

Единственное, что необходимо знать для понимания данной процедуры - функция **stat**

*int stat(const char *file_name, struct stat *buf);*

Эти функции возвращают информацию об указанном файле. Для этого не требуется иметь права доступа к файлу, хотя потребуются права поиска во всех каталогах, указанных в полном имени файла.

stat возвращает информацию о файле **file_name** и заполняет буфер **buf. Istat** идентична **stat**, но в случае символьных ссылок она возвращает информацию о самой ссылке, а не о файле, на который она указывает, **fstat** идентична **stat**, только возвращается информация об открытом файле, на который указывает **filedes** (возвращаемый **open(2))**, а не о **file_name**.

Все эти функции возвращают структуру **stat**, которая содержит следующие поля:

```
struct stat {
dev_t     st_dev;  /* устройство */
ino_t     st_ino;  /* inode */
mode_t st_mode;    /* режим доступа */
nlink_t  st_nlink;    /* количество жестких ссылок */
uid_t     st_uid;  /* идентификатор пользователя-
владельца */
gid_t     st_gid;  /* идентификатор группы-
владельца*/
dev_t     st_rdev;    /* тип устройства */
                      /* (если это устройство) */
off_t     st_size; /* общий размер в байтах */
blksize_t st_blksize;  /* размер блока ввода-вывода
*/
                      /* в файловой системе */
blkcnt_t  st_blocks;   /* количество выделенных
блоков */
time_t   st_atime;    /* время последнего доступа */
time_t   st_mtime;    /* время последней модификации */
```



```
time_t   st_ctime;    /* время последнего изменения */
```

Поле **st_size** задает размер файла (если он обычный или является символьной ссылкой) в байтах. Размер символьной ссылки - длина пути файла на который она ссылается, без конечного NUL.

ЗАКЛЮЧЕНИЕ

Мы попытались создать простейшую системную оболочку. Нам удалось организовать не так уж и мало модулей для работы с файлами и каталогами, которых будет вполне достаточно для работы. Конечно, наш вариант системный оболочки во многом уступает профессиональным средствам. Наш вариант системной оболочки нуждается во многих вещах, например

- Оптимизация
- Дополнение для работы с каталогами
- Создание нормального командного интерпретатора

Но, основная цель- понимание работы системной оболочки.

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. Свободная энциклопедия «Википедия»
2. Русскоязычные руководства Linux
3. http://www.linuxcenter.ru/

Приложение 1

Флаги режима доступа к файлу

Таблица 1

Флаги общего режима

Флаг	Восьмеричное представление	RWX-представление
S_IRWXU	00700	rwx --- ---
S_IRUSR	00400	r-- --- ---
S_IREAD	00400	r-- --- ---
S_IWUSR	00200	-w- --- ---
S_IWRITE	00200	-w- --- ---
S_IXUSR	00100	--x --- ---
S_IEXEC	00100	--x --- ---
S_IRWXG	00070	--- rwx ---
S_IRGRP	00040	--- r-- ---
S_IWGRP	00020	--- -w- ---
S_IXGRP	00010	--- --x ---
S_IRWXO	00007	--- --- rwx
S_IROTH	00004	--- --- r--
S_IWOTH	00002	--- --- -w-
S_IXOTH	00001	--- --- --x

Таблица 2

Флаги расширенного режима

Флаг	Восьмеричное представление	Описание
S_IFMT	0170000	Двоичная маска определения типа файла (побитовое ИЛИ всех следующих ниже флагов)
S_IFDIR	0040000	Каталог
S_IFCHR	0020000	Символьное устройство
S_IFBLK	0060000	Блочное устройство
S_IFREG	0100000	Обычный файл
S_IFIFO	0010000	Канал FIFO
S_IFLNK	0120000	Символическая ссылка

Таблица 3

Дополнительные флаги

Флаг	Восьмеричное представление	Описание
S_ISUID	0004000	Бит SETUID
S_ISGID	0002000	Бит SETGID
S_ISVTX	0001000	Липкий (sticky) бит

Таблица 4

Флаги режима открытия файла

Флаг	Описание
O_RDONLY	Только чтение (0)
O_WRONLY	Только запись (1)
O_RDWR	Чтение и запись(2)
O_CREAT	Создать файл, если не существует
O_TRUNC	Стереть файл, если существует
O_APPEND	Дописывать в конец
O_EXCL	Выдать ошибку, если файл существует при использовании OCREAT
O_DSYNC	Принудительная синхронизация записи
O_RSYNC	Принудительная синхронизация перед чтением
O_SYNC	Принудительная полная синхронизация записи
O_MONBLOCK	Открыть файл в неблокируемом режиме, если это возможно
O_NDELAY	То же, что и O_NONBLOCK
O_NOCTTY	Если открываемый файл - терминальное устройство, не делать его управляющим терминалом процесса
O_NOFOLLOW	Выдать ошибку, если открываемый файл является символической ссылкой
O_DIRECTORY	Выдать ошибку, если открываемый файл не является каталогом
O_DIRECT	Попытаться минимизировать кэширование чтения/записи файла
O_ASYNC	Генерировать сигнал, когда появляется возможность чтения или записи в файл

Приложение 2

Заголовочные файлы

В данном приложении содержится информация по заголовочным файлам наших процедур. Прежде всего опишем, что такое заголовочные файлы.

Заголовочный файл (иногда головной файл, англ. header file),

или подключаемый файл — в языках программирования Си и C++ файл, содержащий определения типов данных, структуры, прототипы функций, перечисления, макросы препроцессора. Имеет по умолчанию расширение .h; иногда для заголовочных файлов языка C++ используют расширение .hpp. Заголовочный файл используется путём включения его текста в данный файл директивой препроцессора **#include.** Чтобы избежать повторного включения одного и того же кода, используются директивы **#ifndef, #define, #endif.**

Заголовочный файл в общем случае может содержать любые конструкции языка программирования, но на практике исполняемый код (за исключением inline-функций в C++) в заголовочные файлы не помещают. Например, идентификаторы, которые должны быть объявлены более чем в одном файле, удобно описать в заголовочном файле, а затем его подключать по мере надобности.

Основная цель использования заголовочных файлов — вынесение описания нестандартных типов и функций за пределы основного файла с кодом. На этом же принципе построены библиотеки: в заголовочном файле перечисляются содержащиеся в библиотеке функции и используемые ею структуры/типы, при этом исходный текст библиотеки может находиться отдельно от текста программы, использующей функции библиотеки или вообще быть недоступным.

Например, по сложившейся традиции, в заголовочных файлах объявляют функции стандартной библиотеки Си и Си++.

Рассмотрим заголовочные файлы наших процедур. Они не содержат ничего, кроме прототипа процедуры и нескольких директив.

Копирование файла
```
#ifndef CP_H
#define CP_H
void copyfile(char *, char *);
#endif    /* CP_H */
```

Удаление файла
```
#ifndef DF_H
#define DF_H
void delfile(char *);
#endif   /* DF_H */
```

Вывод ошибки
```
#ifndef ERR_H
#define ERR_H
#pragma once
void  err1(void);
#endif   /* ERR_H */
```

Создание директории
```
#ifndef MD_H
#define MD_H
void makedir(char *);
#endif /* MD_H */
```

Открытие файла
```
#ifndef OF_H
#define OF_H
void openfile(char *);
#endif   /* OF_H */
```

Удаление директории

```
#ifndef RD_H
#define RD_H
void deldir(char *);
#endif    /* RD_H */
```

Переименование файла

```
#ifndef RF_H
#define RF_H
void renamefile(char *path, char *nname);
#endif  /* RF_H */
```

Информация о файле или директории

```
#ifndef STFILE_H
#define STFILE_H
void statfile(char *path);
#endif /* STFILE_H */
```

Стоит подробнее остановится на директиве **#ifndef**. Эта директива указывает, что нужно компилировать строчки, идущие после неё, если символьная константа не определена (в случае модуля «Информация о файле или директории» символьной константой является **STFILE_H**) и компиляция будет происходить, пока не встретится строчка **#endif**.

Маски прав доступа

Маска прав доступа определяет кто имеет доступ к информации. Следует разделять маски для файлов и для каталогов, они различаются на один бит-Т бит. Если Т бит установлен в 1, то удалять файл из директории имеет право только владелец директории/ иначе(то есть если стоит 0) -удалять может не только владелец. Рассмотрим теперь оставшиеся три бита маски (они одинаковы для директории и для файла).

400 — владелец имеет право на чтение;
200 — владелец имеет право на запись;
100 — владелец имеет право на выполнение;
40 — группа имеет право на чтение;
20 — группа имеет право на запись;
10 — группа имеет право на выполнение;
4 — остальные имеют право на чтение;
2 — остальные имеют право на запись;
1 — остальные имеют право на выполнение.

Суммировав эти коды можно получить символьную запись. Например, 444:

400+40+4=444 — все имеют право только на чтение.