Министерство образования и науки Российской Федерации

«Томский государственный университет систем управления и радиоэлектроники»

(ТУСУР)

УТВЕРЖДАЮ

Проректор по учебной работе

_____ Л.А. Боков

«____» _____ 2011 г.

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ К ПРАКТИЧЕСКИМ ЗАНЯТИЯМ

по дисциплине

«Архитектура систем на кристалле»

Составлена кафедрой «Управление инновациями»

Для студентов, обучающихся

по направлению подготовки 220600 «Инноватика»

Магистерская программа «Мультимедийные многопроцессорные системы на кристалле»

Форма обучения очная

Составитель

Ассистент _____ Н.В. Милованов

Томск 2011

Практическое занятие № 1. Знакомство с ОС GNU/Linux. Вход в систему. Запуск программ.

Цель работы.

Получить представление о типичном ходе работы с ОС GNU/Linux, познакомиться с интерфейсом, научиться самостоятельно решать возникающие проблемы. Познакомиться с принципами перенаправления стандартных потоков данных, научиться пользоваться конвейером.

Теория.

Для входа в систему необходимо ввести имя пользователя, а потом, по запросу, и пароль для входа в систему. Если это первый вход в систему после ее установки, то входить надо под именем "root". Это единственный пользователь, для которого обязательно заводится счет или бюджет (account) во время инсталляции. Этот пользователь является полным хозяином системы (как сейчас, так и в последующем), то есть имеет неограниченный доступ к ее ресурсам, может заводить и удалять других пользователей, останавливать систему и т. д. Неосторожное поведение пользователя с такими правами легко может привести к печальным последствиям, вплоть до полного краха системы. Поэтому обычно под этим именем входят в систему только для выполнения административных задач. Но у нас сейчас не такой случай, так что в ответ на приглашение `login`**:** вводим полученное от администратора имя и нажимаем клавишу <Enter> (или <Return>). Система выдаст запрос на ввод пароля:

Password:

Очевидно, что в ответ надо вводить пароль того пользователя, имя которого было введено ранее. Заметим, что если после ввода имени очень долго не вводить пароль, то система снова вернется к запросу имени пользователя. После ввода пароля вы увидите примерно такую надпись:

[root@localhost /root]#

Такая строка называется приглашением. Появление приглашения означает, что система готова воспринять и выполнить вашу команду. Сейчас это свидетельствует о том, что вы успешно вошли в систему. Вы видите черный экран и приглашение системы к вводу команды — то, что в MS-DOS или Windows принято называть режимом командной строки. Мы будем называть этот режим текстовым (в отличие от графического режима, предоставляемого системой X Window).

В приведенном примере приглашение включает в себя указание имени пользователя (`root`), имени системы (`localhost`) и текущего каталога (`/root`). Вид приглашения тоже можно изменить. Во всех последующих примерах мы будем использовать приглашение, состоящее только из имени пользователя.

Прежде чем предложить вам ввести первую команду, надо сказать, что в любой UNIX-системе учитывается регистр символов, т. е. различаются строчные и прописные буквы. Поэтому вводить все команды и их параметры следует именно так, как указано в примерах, учитывая регистр.

Первая команда, о которой нужно знать каждому пользователю любой UNIX-системы — это команда `man`. Команда `man` — это система встроенной помощи системы Linux. Вводить ее надо с параметром — именем другой команды или ключевым словом, например,

[root]# man passwd

В ответ вы получите описание соответствующей команды или информацию по теме, обозначенной ключевым словом. Поскольку информация обычно не помещается на одном экране, при просмотре можно пользоваться клавишами <PageUp> и <PageDown>, а также клавишей пробела. Нажатие клавиши <Q> в любой момент приводит к выходу из режима просмотра и возврату в режим ввода команд. Попробуйте просмотреть информацию по рассмотренным уже

командам `login` и `passwd`. Заметим, что точно также можно получить информацию по самой команде `man`. Введите

[root]# man man

К сожалению, в большинстве случаев информация выдается по-английски. Если вы не читаете по-английски, то терпеливо читайте настоящее руководство или другую подходящую книгу по данной теме.

Вы можете попробовать вводить еще некоторые команды и понаблюдать за реакцией системы. Попробуйте, например, команды, перечисленные в следующей таблице (вводите их с приведенными в таблице параметрами).

Команда	Краткое описание
whoami	Сообщает имя, с которым вы вошли в систему в данном сеансе работы
w или who	Сообщает, какие пользователи работают в данный момент в системе
pwd	Сообщает имя текущего каталога
ls -l	Выдает список файлов и подкаталогов текущего каталога
cd <имя_каталога>	Осуществляет смену текущего каталога
ps ax	Выдает список выполняющихся процессов

Просмотрите описания этих команд с помощью команды `man`.

Итак, вы приобрели первый опыт работы в текстовом, или "консольном", режиме системы Linux. Понятия "терминала" и "консоли", которые встретятся нам еще не раз, требуется, вероятно, дополнительно пояснить.

Когда создавалась система UNIX, компьютеры были большими (мэйнфреймами), и пользователи работали на них через множество последовательных интерфейсов для подключения удаленных терминалов. Терминал — это устройство, которое предназначено для взаимодействия

пользователя с компьютером и состоит из монитора и клавиатуры. К вашему персональному компьютеру наверняка не подключены удаленные терминалы, но есть клавиатура и монитор, которые и выполняют роль терминала пользователя (только в его состав добавилась мышь).

У мэйнфреймов имелся особый терминал, который предназначался для системного администратора и назывался консолью. Консоль обычно подсоединялась к компьютеру не по последовательному интерфейсу, а через отдельные разъемы (иногда в качестве устройства вывода в ее состав вместо монитора входило печатающее устройство).

Поскольку в UNIX-системах обычно соблюдаются традиции, клавиатура и монитор персонального компьютера ведут себя так же, как ранее консоль. Преимущество такого решения состоит в том, что все старые программы, создававшиеся для администраторов UNIX, без проблем работают и на новом типе системной консоли.

Но, кроме консоли, Linux позволяет подключать к компьютеру и удаленные терминалы и, более того, обеспечивает возможность работы с несколькими виртуальными терминалами с одной консоли. Нажмите комбинацию клавиш <Ctrl>+<Alt>+<F2>. Вы снова увидите приглашение `login`**:**. Однако это не возврат к началу работы с системой — вы просто переключились в другой виртуальный терминал. Здесь вы можете зарегистрироваться под другим именем. Попробуйте войти в систему под своим именем. После этого нажмите комбинацию клавиш <Ctrl >+< Alt>+<F1>. Вы вернетесь к первому экрану. По умолчанию Red Hat Linux открывает при запуске 6 параллельных сеансов работы (виртуальных терминалов), и этим иногда очень удобно пользоваться. Для переключения между виртуальными терминалами используются комбинации <Ctrl >+< Alt >+<F1> — <Ctrl>+<Alt>+<F6>. (Заметим, что при работе в текстовом режиме тот же результат можно получить, используя комбинации <Alt >+<

F1> — <Alt >+< F6>, однако в графическом режиме без клавиши <Ctrl> не обойтись, так что лучше сразу привыкать к комбинациям из 3 клавиш). Кстати, если в процессе работы вы забыли, в каком терминале находитесь в данный момент, воспользуйтесь командой `tty`, которая выводит имя терминала в следующем формате: /dev/tty2.

Сразу же скажем, что, если вы хотите завершить сеанс работы с системой в одном из терминалов, вы можете сделать это нажатием комбинации клавиш <Ctrl >+< D>. Это не приведет ни к остановке работы компьютера, ни к перезагрузке системы. Не забывайте, что Linux — многозадачная и многопользовательская система. Завершение работы одного пользователя не означает, что надо выключать компьютер. Просто завершается сеанс работы одного из пользователей, и система снова выводит в данном терминале приглашение, которое вы уже видели. Можно завершить сеанс работы, введя одну из команд `logout` или `exit`.

Зная теперь, как открыть и закрыть сеанс работы в системе, выполните приведенные выше рекомендации, т. е. заведите себя как рядового пользователя (без суперпользовательских прав), завершите все сеансы работы, открытые от имени `root`, и снова войдите в систему под своим новым именем.

Теперь надо сказать несколько слов об оболочке. Оболочка, или просто shell (это слово часто не переводят, а оставляют в английском написании), — это программа, которая осуществляет все общение с пользователем. Именно оболочка воспринимает все команды, вводимые пользователем с клавиатуры, и организует исполнение этих команд. Поэтому оболочку можно назвать еще командным процессором. Строго говоря, когда выше говорилось, например, "система выводит приглашение", это неправильно, поскольку приглашение выводит именно оболочка, ожидая ввода пользователем очередной команды. Каждый раз, когда очередной пользователь входит в систему, команда

login запускает для него командный процессор — оболочку. Если вы логировались со второго терминала под именем пользователя jim (или под другим выбранным вами именем), то обратите теперь внимание на различие в приглашениях у пользователей root и jim. У пользователя root приглашение оканчивается символом #, а у всех остальных пользователей — символом $.

Оболочку может запускать не только команда login. Вы можете просто ввести команду bash (именно так называется программа-оболочка в системе Red Hat Linux) и тем самым запустить новый экземпляр оболочки. Выходя из него (по команде exit или по комбинации клавиш <Ctrl >+< D>) вы вернетесь к предыдущему экземпляру оболочки.

Оболочка bash является не только командным процессором, но и мощным языком программирования. В ней имеется целый ряд встроенных (внутренних) команд и операторов, а, кроме того, в качестве команды может использоваться любая программа, хранящаяся в виде файла на диске. Список встроенных команд можно получить по команде help. Попробуйте! Детальную информацию по конкретной встроенной команде выдает та же команда help с указанием в качестве параметра имени встроенной команды, например: help cd. Стоит отметить, что для UNIX-подобных систем разработано несколько альтернативных bash оболочек. Их можно использовать и в Linux, но по умолчанию запускается именно bash.

Рассмотрим теперь еще одну команду, которую вам необходимо знать. Как уже было сказано выше, входить в систему под именем суперпользователя не рекомендуется, поскольку любое неосторожное действие суперпользователя может привести к нежелательным последствиям. Входя под именем простого пользователя, вы, по крайней мере, не можете по неосторожности удалить или испортить системные файлы. В то же время, имеется ряд действий

(например, монтирование файловых систем), выполнить которые может только суперпользователь. Именно в таких ситуациях выручает команда `su`. Достаточно ввести команду `su` и текущая оболочка запустит для вас новый экземпляр оболочки, в который вы попадете уже с правами пользователя `root`. Естественно, что для этого вам придется (в ответ на соответствующий запрос) ввести пароль этого пользователя. Закончив выполнять администраторские действия, выйдите из оболочки, и вы снова станете непривилегированным пользователем с отведенными ему полномочиями.

Если вы вошли в систему под именем `root`, то вы можете аналогичным образом запустить новый экземпляр оболочки от имени любого пользователя, пароль которого вы знаете. Но для этого надо указать имя этого пользователя в командной строке, например:

[user]$ su jim

Когда мы вводим su без указания имени, по умолчанию подставляется имя суперпользователя root.

Задание на выполнение.

1. Войти в систему под своей учетной записью.
2. Узнать, в какие группы входит пользователь, версию ядра.
3. Определить, какие права имеет пользователь на файл /dev/ttyS0
4. Подсчитать количество выполняющихся процессов. Решить простое арифметическое выражение с помощью команды bc.
5. Сохранить справку о команде grep в текстовый файл.

Практическое занятие № 2. Командная оболочка bash. Создание и выполнение скриптов.

Цель работы.

Научиться пользоваться командной оболочкой bash. Разобраться с принципом работы переменных окружения.

Теория.

Командный язык shell (в переводе - оболочка) фактически есть язык программирования очень высокого уровня. На этом языке пользователь осуществляет управление компьютером. Обычно, после входа в систему вы начинаете взаимодействовать с командной оболочкой (если угодно - она начинает взаимодействовать с вами). Признаком того, что оболочка (shell) готова к приему команд, служит выдаваемый ею на экран промптер (символ-приглашение). В простейшем случае это знак доллар ("$").

Обратите внимание: shell - это одна из многих команд UNIX. То есть в набор команд оболочки (интерпретатора) "shell" входит команда "sh" - вызов интерпретатора "shell". Первый "shell" вызывается автоматически при вашем входе в систему и выдает на экран промптер. После этого вы можете вызывать на выполнение любые команды, в том числе и снова сам "shell", который вам создаст новую оболочку внутри прежней.

Команды в shell обычно имеют следующий формат:

<имя команды> <флаги> <аргумент(ы)>

Например:

```
ls  -ls  /usr/bin
```

ls	имя команды выдачи содержимого директория,
-ls	флаги ("-" - признак флагов, l - длинный формат, s - об'ем файлов в блоках).
/usr/bin	директорий, для которого выполняется команда.

Эта команда выдаст на экран в длинном формате содержимое директория /usr/bin, при этом добавит информацию о размере каждого файла в блоках.

К сожалению, такая структура команды выдерживается далеко не всегда. Не всегда перед флагами ставится минус, не всегда флаги идут одним словом. Есть разнообразие и в представлении аргументов. К числу команд, имеющих экзотические форматы, относятся и такие "ходовые" команды, как cc, tar, dd, find и ряд других.

Как правило (но не всегда), первое слово (т.е. последовательность символов до пробела, табуляции или конца строки) shell воспринимает, как команду. Поэтому в командной строке

```
cat cat
```

первое слово будет расшифровано shell, как команда (конкатенации), которая выдаст на экран файл с именем "cat" (второе слово), находящийся в текущем директории.

Управление заданиями в Bash

Одна из наиболее мощных возможностей Bash - это возможность помочь пользователю в управлении исполнением различных команд. Каждая команда имеет стандартный механизм для обработки ввода-вывода:

STDIN (стандартное устройство ввода) позволяет программе получать входные данные из командной оболочки.

STDOUT (стандартное устройство вывода) позволяет программе передавать свои выходные данные командной оболочке.

STDERR (стандартное устройство вывода ошибок) позволяет программе передавать командной оболочке информацию о возникающих ошибках.

Обычно STDIN - это клавиатура, с помощью которой передается информация команде (как это делается в строке приглашения Bash), а STDOUT и STDERR - это экран, на который выводятся результаты. Однако можно изменить этот механизм ввода-вывода, например, так, чтобы команда читала и записывала данные в файл.

Для изменения стандартного механизма ввода-вывода в Bash используются команды перенаправления:

Последовательность	Описание
command < file	перенаправляет STDIN на чтение из файла.
command > file	перенаправляет STDOUT на запись в файл.
command >> file	перенаправляет STDOUT на дозапись в файл.
command 2> file	перенаправляет STDERR на запись в файл.
command1 | command2	подсоединяет STDOUT команды *command1* к STDIN команды *command2*.

Последний пункт в таблице известен как *конвейер* и используется для объединения команд. Объединение команд с помощью конвейера - это ключевая техника программирования в командной оболочке UNIX, использующаяся, как продемонстрировано на рисунке 2. В этой модели каждая команда выполняет одну простую задачу и передает результат своей работы другой команде, которая также выполняет простую задачу, и так далее. Например, можно с помощью программы cat передать через конвейер содержимое файла команде grep, которая через конвейер передает все найденные строки команде wc, которая подсчитывает и выводит на экран количество строк в файле, содержащих заданную подстроку.

Асинхронное выполнение

Исполнение команд, которое описывалось выше, является синхронным, то есть в каждый момент времени выполняется только одна программа. Иногда команда или программа должна работать в течение долгого времени. Чтобы не отказываться от интерактивного использования командной оболочки, можно выполнять команды асинхронно. Для этого нужно дополнить команду знаком амперсанда (&). Это указывает командной оболочке запустить команду в фоновом режиме, что позволит продолжить работу в Bash. Пример демонстрирует эту и другие техники управления заданиями.

```
rb$ grep paper.pdf /var/log/httpd/access.log | wc -l
5
rb$ python demo.py &
[1] 20451
rb$ jobs
[1]+  Running                 python demo.py &
rb$ fg 1
python demo.py
```

В первом примере команда grep ищет строку *paper.pdf* в лог-файле Web-сервера Apache Web. Результат работы этой команды передается через конвейер команде wc -l, которая считает количество строк. Таким образом, целиком команда подсчитывает число строк, содержащих paper.pdf в лог-файле.

Во втором примере программа на языке Python запускается в фоне для долговременной работы. Bash запускает это задание асинхронно в фоне и указывает идентификатор задания. Используя команду jobs, можно просмотреть список всех запущенных команд. В этом примере запущена только одна программа, она выводится из фоновой работы в синхронное выполнение с помощью команды fg.

Задание на выполнение.

1. Написать скрипт, который выводит список переданных ему аргументов в виде таблицы.
2. Написать скрипт, который передает часть аргументов другому скрипту через переменные окружения.
3. Написать скрипт, который выполняет собранное в нестандартной директории приложение с зависимостями, при этом дублирует функционал инициализационного скрипта (/etc/rc.d).

Практическое занятие № 3. Работа с Embedded Linux на примере STB820. Сравнение с GNU/Linux для x86.

Цель работы.

Освоить навыки подключения ЦТП к компьютеру для последующего обмена информацией между ними, запуска и отладки программного обеспечения на ЦТП.

Задание на выполнение.

1. Подключиться к консоли ЦТП с помощью COM-порта. Настроить программный эмулятор терминала.
2. Настроить на ЦТП подключение к сети. Подключиться к ЦТП с помощью telnet.
3. Изучить лог загрузки ЦТП, сравнить с загрузочными скриптами.
4. Изучить набор программного обеспечения на встроенной флэш-памяти ЦТП.
5. Подключить флэш-накопитель к ЦТП, вручную смонтировать во временную директорию, скопировать на флэш настройки ЦТП.
6. Настроить NFS сервер на рабочем компьютере. Вручную смонтировать NFS на ЦТП.
7. Настроить Samba сервер на рабочем компьютере. Смонтировать вручную общую папку, скопировать на нее настройки ЦТП.

Практическое занятие № 4. Реализация простого приложения для GNU/Linux.

Цель работы.
Научиться пользоваться инструментами Linux для создания приложений.

Теория.

Знакомство с компилятором GCC

Средствами, традиционно используемыми для создания программ для открытых операционных систем, являются инструменты разработчика GNU. Справка. Проект GNU был основан в 1984 году Ричардом Столлманом. Его необходимость была вызвана тем, что в то время сотрудничество между программистами было затруднено, так как владельцы коммерческого программного обеспечения чинили многочисленные препятствия такому сотрудничеству. Целью проекта GNU было создание комплекта программного обеспечения под единой лицензией, которая не допускала бы возможности присваивания кем-то эксклюзивных прав на это ПО. Частью этого комплекта и является набор инструментов для разработчика, которым мы будем пользоваться, и который должен входить во все дистрибутивы Linux.

Одним из этих инструментов является компилятор GCC. Первоначально эта аббревиатура расшифровывалась, как GNU C Compiler. Сейчас она означает GNU Compiler Collection.

Создадим первую программу с помощью GCC. По сложившейся традиции первая программа будет просто выводить в консоли приветствие «Hello world!» – «Здравствуй Мир!».

Файлы с исходными кодами программ, которые мы будем создавать, это обычные текстовые файлы, и создавать их можно с помощью любого текстового редактора (например, GEdit KWrite, Kate, а также более

традиционные для пользователей Linux – vi и emacs). Помимо текстовых редакторов, существуют специализированные среды разработки со своими встроенными редакторами. Одним из таких средств является KDevelop. Интересно, что в нём есть встроенный редактор и встроенная консоль, расположенная прямо под редактором. Так что можно прямо в одной программе, не переключаясь между окнами, и редактировать код, и давать консольные команды.

Создайте отдельный каталог hello. Это будет каталог нашего первого проекта. В нём создайте текстовый файл hello.c со следующим текстом:

```
#include <stdio.h>

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

Затем в консоли зайдите в каталог проекта. Наберите команду

```
gcc hello.c
```

Теперь посмотрите внимательно, что произошло. В каталоге появился новый файл a.out. Это и есть исполняемый файл. Запустим его. Наберите в консоли:

```
./a.out
```

Программа должна запуститься, то есть должен появиться текст:

```
Hello world!
```

Компилятор gcc по умолчанию присваивает всем созданным исполняемым файлам имя a.out. Если хотите назвать его по-другому, нужно к команде на компиляцию добавить флаг -o и имя, которым вы хотите его назвать. Давайте наберём такую команду:

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

Мы видим, что в каталоге появился исполняемый файл с названием hello. Запустим его.

./hello

Как видите, получился точно такой же исполняемый файл, только с удобным для нас названием.

Флаг -o является лишь одним из многочисленных флагов компилятора gcc. Некоторые другие флаги мы рассмотрим позднее. Чтобы просмотреть все возможные флаги, можно воспользоваться справочной системой man. Наберите в командной строке:

man gcc

Перед вами предстанет справочная система по этой программе. Просмотрите, что означает каждый флаг. С некоторыми из них мы скоро встретимся. Выход из справочной системы осуществляется с помощью клавиши q.

Вы, конечно, обратили внимание, что, когда мы запускаем программу из нашего каталога разработки, мы перед названием файла набираем точку и слэш. Зачем же мы это делаем?

Дело в том, что, если мы наберём только название исполняемого файла, операционная система будет искать его в каталогах /usr/bin и /usr/local/bin, и, естественно, не найдёт. Каталоги /usr/bin и /usr/local/bin – системные каталоги размещения исполняемых программ. Первый из них предназначен для размещения стабильных версий программ, как правило, входящих в дистрибутив Linux. Второй – для программ, устанавливаемых самим пользователем (за стабильность которых никто не ручается). Такая система нужна, чтобы отделить их друг от друга. По умолчанию при сборке программы устанавливаются в каталог /usr/local/bin. Крайне нежелательно помещать что-либо лишнее в /usr/bin или удалять что-то оттуда вручную, потому что это может привести к краху системы. Там должны размещаться программы, за стабильность которых отвечают разработчики дистрибутива.

Чтобы запустить программу, находящуюся в другом месте, надо прописать полный путь к ней, например так:

/home/dima/projects/hello/hello

Или другой вариант: прописать путь относительно текущего каталога, в котором вы в данной момент находитесь в консоли. При этом одна точка означает текущий каталог, две точки – родительский. Например, команда ./hello запускает программу hello, находящуюся в текущем каталоге, команда ../hello – программу hello, находящуюся в родительском каталоге, команда ./projects/hello/hello – программу во вложенных каталогах, находящихся внутри текущего.

Есть возможность добавлять в список системных путей к программам дополнительные каталоги. Для этого надо добавить новый путь в системную переменную PATH.

Теперь рассмотрим, что делает программа gcc. Её работа включает три этапа: обработка препроцессором, компиляция и компоновка (или линковка). Препроцессор включает в основной файл содержимое всех заголовочных файлов, указанных в директивах #include. В заголовочных файлах обычно находятся объявления функций, используемых в программе, но не определённых в тексте программы. Их определения находятся где-то в других файлах с исходным кодом или в бинарных библиотеках.

Вторая стадия – компиляция. Она заключается в превращении текста программы на языке C/C++ в набор машинных команд. Результат сохраняется в объектном файле. Разумеется, на машинах с разной архитектурой процессора двоичные файлы получаются в разных форматах, и на одной машине невозможно запустить бинарник, собранный на другой машине (разве только, если у них одинаковая архитектура процессора и одинаковые операционные системы). Вот почему программы для UNIX-подобных систем распространяются в виде исходных кодов: они должны

быть доступны всем пользователям, независимо от того, у кого какой процессор и какая операционная система.

Последняя стадия – компоновка. Она заключается в связывании всех объектных файлов проекта в один, связывании вызовов функций с их определениями и присоединении библиотечных файлов, содержащих функции, которые вызываются, но не определены в проекте. В результате формируется запускаемый файл – наша конечная цель. Если какая-то функция в программе используется, но компоновщик не найдёт место, где эта функция определена, он выдаст сообщение об ошибке, и откажется создавать исполняемый файл.

Теперь посмотрим на практике, как всё это выглядит. Напишем другую программу. Это будет простой калькулятор, способный складывать, вычитать, умножать и делить. При запуске он будет запрашивать по очереди два числа, над которыми следует произвести действие, а затем потребует ввести знак арифметического действия. Это могут быть четыре знака: «+», «–», «*», «/». После этого программа выводит результат и останавливается (возвращает нас в операционную систему, а точнее – в командный интерпретатор, из которого мы программу и вызывали).

Создадим для проекта новую папку kalkul, в ней создадим файл kalkul.c.

```
#include <stdio.h>
int main(void)
{
float num1;
float num2;
char op;
printf("Первое число: ");
scanf("%f",&num1);
printf("Второе число: ");
```

```
scanf("%f",&num2);
printf("Оператор ( + - * / ): ");
while ((op = getchar()) != EOF)
{
if (op == '+')
{
printf("%6.2f\n",num1 + num2);
break;
}
else if(op == '-')
{
printf("%6.2f\n",num1 - num2);
break;
}
else if(op == '*')
{
printf("%6.2f\n",num1 * num2);
break;
}
else if(op == '/')
{
if(num2 == 0)
{
printf("Ошибка: деление на ноль!\n");
break;
}
else
{
```

```
printf("%6.2f\n",num1 / num2);
break;
}
}
}
return 0;
}
```

Итак, сначала выполняется препроцессинг. Для того чтобы посмотреть, что на этом этапе делается, воспользуемся опцией -E. Она останавливает выполнение программы на этапе обработки препроцессором. В результате получается файл исходного кода с включённым в него содержимым заголовочных файлов.

В нашем случае мы включали один заголовочный файл – stdio.h – коллекцию стандартных функций ввода-вывода. Эти функции и выводили на консоль нужный текст, а также считывали с консоли вводимые нами слова.

Введите следующую команду:

```
gcc -E kalkul.c -o kalkul.cpp
```

Полученному файлу мы дали имя kalkul.cpp. Откройте его. Обратите внимание на то, что он весьма длинный. Это потому, что в него вошёл весь код заголовочного файла stdio.h. Кроме того, препроцессор сюда добавил некоторые теги, указывающие компилятору способ связи с объявленными функциями. Основной текст нашей программы виден только в самом низу.

Можете заодно посмотреть, какие ещё функции объявлены в заголовочном файле stdio.h. Если вам захочется получить информацию о какой-нибудь функции, можно поинтересоваться о ней во встроенном руководстве man. Например, чтобы узнать, что делает функция fopen, можно набрать:

```
man fopen
```

Много информации также есть в справочной системе info.

info fopen

Можно поинтересоваться и всем заголовочным файлом сразу.

man stdio.h

info stdio.h

Посмотрим теперь следующий этап. Создадим объектный файл. Объектный файл представляет собой «дословный» перевод нашего программного кода на машинный язык, пока без связи вызываемых функций с их определениями. Для формирования объектного файла служит опция -c.

gcc -c kalkul.c

Название получаемого файла можно не указывать, так как компилятор просто берёт название исходного и меняет расширение .c на .o (указать можно, если нам захочется назвать его по-другому).

Если мы создаём объектный файл из исходника, уже обработанного препроцессором (например, такого, какой мы получили выше), то мы должны обязательно указать явно, что компилируемый файл является файлом исходного кода, обработанный препроцессором, и имеющий теги препроцессора. В противном случае он будет обрабатываться, как обычный файл C++, без учёта тегов препроцессора, а значит, связь с объявленными функциями не будет устанавливаться. Для явного указания на язык и формат обрабатываемого файла служит опция -x. Файл C++, обработанный препроцессором, обозначается cpp-output.

gcc -x cpp-output -c kalkul.cpp

Наконец, последний этап – компоновка. Получаем из объектного файла исполняемый.

gcc kalkul.o -o kalkul

Можно его запускать.

./kalkul

Поясним, зачем нужны промежуточные этапы. Дело в том, что настоящие программы очень редко состоят из одного файла. Как правило, исходных файлов несколько, и они объединены в проект. А в некоторых случаях программу приходится компоновать из нескольких частей, написанных на разных языках. Тогда приходится запускать компиляторы разных языков, чтобы каждый получил объектный файл из своего исходника, а затем уже эти полученные объектные файлы компоновать в исполняемую программу.

Пример проекта из нескольких файлов

Напишем теперь программу, состоящую из двух исходных файлов и одного заголовочного. Для этого возьмём наш калькулятор и переделаем его. Теперь после введения первого числа надо сразу вводить действие. Если действие оперирует только с одним числом (как в случае синуса, косинуса, тангенса, квадратного корня), результат сразу будет выведен. Если понадобится второе число, оно будет специально запрашиваться.

Создадим каталог проекта kalkul2. В нём создадим три файла: calculate.h, calculate.c, main.c.

Файл calculate.h:

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

#ifndef CALCULATE_H_
#define CALCULATE_H_

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

#endif /*CALCULATE_H_*/
```

Файл calculate.c:

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

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

float Calculate(float Numeral, char Operation[4])
{
float SecondNumeral;
if(strncmp(Operation, "+", 1) == 0)
{
printf("Второе слагаемое: ");
scanf("%f",&SecondNumeral);
return(Numeral + SecondNumeral);
}
else if(strncmp(Operation, "-", 1) == 0)
{
printf("Вычитаемое: ");
scanf("%f",&SecondNumeral);
return(Numeral - SecondNumeral);
}
else if(strncmp(Operation, "*", 1) == 0)
{
printf("Множитель: ");
scanf("%f",&SecondNumeral);
return(Numeral * SecondNumeral);
```

```
}
else if(strncmp(Operation, "/", 1) == 0)
{
printf("Делитель: ");
scanf("%f",&SecondNumeral);
if(SecondNumeral == 0)
{
printf("Ошибка: деление на ноль! ");
return(HUGE_VAL);
}
else
return(Numeral / SecondNumeral);
}
else if(strncmp(Operation, "pow", 3) == 0)
{
printf("Степень: ");
scanf("%f",&SecondNumeral);
return(pow(Numeral, SecondNumeral));
}
else if(strncmp(Operation, "sqrt", 4) == 0)
return(sqrt(Numeral));

else if(strncmp(Operation, "sin", 3) == 0)
return(sin(Numeral));
else if(strncmp(Operation, "cos", 3) == 0)
return(cos(Numeral));
else if(strncmp(Operation, "tan", 3) == 0)
return(tan(Numeral));
```

```
else
{
printf("Неправильно введено действие ");
return(HUGE_VAL);
}
}
```

Файл main.c:

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

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

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

У нас есть два файла исходного кода (c-файлы) и один заголовочный (h-файл). Заголовочный файл включается в оба c-файла.

Скомпилируем calculate.c.

gcc -c calculate.c

Получили calculate.o. Затем main.c.

gcc -c main.c

И вот main.o перед нами! Теперь надо из этих двух объектных файлов сделать запускаемый.

gcc calculate.o main.o -o kalkul

Не получилось! Вместо запускаемого файла в консоли появилось:

calculate.o(.text+0x1b5): In function `Calculate':

calculate.c: undefined reference to `pow'

calculate.o(.text+0x21e):calculate.c: undefined reference to `sqrt'

calculate.o(.text+0x274):calculate.c: undefined reference to `sin'

calculate.o(.text+0x2c4):calculate.c: undefined reference to `cos'

calculate.o(.text+0x311):calculate.c: undefined reference to `tan'

collect2: ld returned 1 exit status

Давайте разберёмся. Undefined reference означает ссылку на функцию, которая не определена. В данном случае gcc не нашёл определения функций pow, sqrt, sin, cos, tan. Где их взять? Как уже говорилось выше, определения функций могут находиться в библиотеках. Это скомпилированные двоичные файлы, содержащие коллекции однотипных операций, которые часто вызываются из многих программ, а потому нет смысла многократно писать их код в программах. Стандартное расположение файлов библиотек – каталоги /usr/lib и /usr/local/lib (при желании можно добавить путь). Если библиотечный файл имеет расширение .a, то это статическая библиотека, то есть при компоновке весь её двоичный код включается в исполняемый файл. Если расширение .so, то это динамическая библиотека. Это значит, что в исполняемый файл программы помещается только ссылка на библиотечный файл, а уже из него и запускается функция.

Когда мы писали программу hello, мы использовали функцию printf для вывода текстовой строки. Однако мы нигде не писали определения этой функции. Откуда же она тогда вызывается?

Просто при компоновке любой программы компилятор gcc по умолчанию включает в запускаемый файл библиотеку libc. Это стандартная библиотека языка C. Она содержит рутинные функции, необходимые абсолютно во всех программах, написанных на C, в том числе и функцию printf. Поскольку библиотека libc нужна во всех программах, она включается по умолчанию, без необходимости давать отдельное указание на её включение.

Остальные библиотеки надо требовать включать явно – только те, которые реально нужны. Нам в данном случае нужна библиотека libm. Именно она содержит все основные математические функции. Она требует включения в текст программы заголовочного файла <math.h>.

Помимо этого дистрибутивы Linux содержат и другие библиотеки, например:

libGL - трёхмерная графика в стандарте OpenGL. Требуется заголовочный файл <GL/gl.h>.

libcrypt Криптографические функции. Требуется заголовочный файл <crypt.h>.

libcurses Псевдографика в символьном режиме. Требуется заголовочный файл <curses.h>.

libform Создание экранных форм в текстовом режиме. Требуется заголовочный файл <form.h>.

libgthread Поддержка многопоточного режима. Требуется заголовочный файл <glib.h>.

libgtk Графическая библиотека в режиме X Window. Требуется заголовочный файл <gtk/gtk.h>.

libhistory Работы с журналами. Требуется заголовочный файл <readline/readline.h>.

libjpeg Работа с изображениям в формате JPEG. Требуется заголовочный файл <jpeglib.h>.

libncurses Работа с псевдографикой в символьном режиме. Требуется заголовочный файл <ncurses.h>.

libpng Работа с графикой в формате PNG. Требуется заголовочный файл <png.h>.

libpthread Многопоточная библиотека POSIX. Стандартная многопоточная библиотека для Linux. Требуется заголовочный файл <pthread.h>.

libreadline Работа с командной строкой. Требуется заголовочный файл <readline/readline.h>.

libtiff Работа с графикой в формате TIFF. Требуется заголовочный файл <tiffio.h>.

libvga Низкоуровневая работа с VGA и SVGA. Требуется заголовочный файл <vga.h>.

А также многие-многие другие.

Обратите внимание, что названия всех этих библиотек начинаются с буквосочетания lib-. Для их явного включения в исполняемый файл, нужно добавить к команде gcc опцию -l, к которой слитно прибавить название библиотеки без lib-. Например, чтобы включить библиотеку libvga, надо указать опцию -lvga.

Нам нужны математические функции pow, sqrt, sin, cos, tan. Они, как уже было сказано, находятся в математической библиотеке libm. Следовательно, чтобы подключить эту библиотеку, мы должны указать опцию -lm.

```
gcc calculate.o main.o -o kalkul -lm
```

Наконец запускаемый файл создан!

Make-файлы

У вас, вероятно, появился вопрос: можно ли не компилировать эти файлы по отдельности, а собрать сразу всю программу одной командой? Можно.

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

Это удобно для нашей небольшой программы, потому что она состоит всего из двух с-файлов. Однако профессиональная программа может состоять из нескольких десятков таких файлов. Каждый раз набирать названия их всех в одной строке было бы делом чрезмерно утомительным. Но есть возможность решить эту проблему. Названия всех исходных файлов и все команды для сборки программы можно поместить в отдельный текстовый файл. А потом считывать их оттуда одной короткой командой.

Давайте создадим такой текстовый файл и воспользуемся им. В каталоге проекта kalkul2 удалите все файлы, кроме calculate.h, calculate.c, main.c. Затем создайте в этом же каталоге новый файл, назовите его Makefile (без расширений). Поместите туда следующий текст.

```
kalkul: calculate.o main.o
	gcc calculate.o main.o -o kalkul -lm
calculate.o: calculate.c calculate.h
	gcc -c calculate.c
main.o: main.c calculate.h
	gcc -c main.c
clean:
	rm -f kalkul calculate.o main.o
install:
	cp kalkul /usr/local/bin/kalkul
uninstall:
	rm -f /usr/local/bin/kalkul
```

Обратите внимание на строки, введённые с отступом от левого края. Этот отступ получен с помощью клавиши Tab. Только так его и надо делать! Если будете использовать клавишу «Пробел», команды не будут исполняться.

Затем дадим команду, состоящую всего из одного слова:

make

И сразу же в нашем проекте появляются и объектные файлы, и запускаемый файл. Программа make как раз и предназначена для интерпретации команд,

находящихся в файле со стандартным названием Makefile. Рассмотрим его структуру.

Makefile является списком правил. Каждое правило начинается с указателя, называемого «Цель». После него стоит двоеточие, а далее через пробел указываются зависимости. В нашем случае ясно, что конечный файл kalkul зависит от объектных файлов calculate.o и main.o. Поэтому они должны быть собраны прежде сборки kalkul. После зависимостей пишутся команды. Каждая команда должна находиться на отдельной строке, и отделяться от начала строки клавишей Tab. Структура правила Makefile может быть очень сложной. Там могут присутствовать переменные, конструкции ветвления, цикла. Этот вопрос требует отдельного подробного изучения.

Если мы посмотрим на три первых правила, то они нам хорошо понятны. Там те же самые команды, которыми мы уже пользовались. А что же означают правила clean, install и uninstall?

В правиле clean стоит команда rm, удаляющая исполняемый и объектные файлы. Флаг -f означает, что, если удаляемый файл отсутствует, программа должна это проигнорировать, не выдавая никаких сообщений. Итак, правило clean предназначено для «очистки» проекта, приведения его к такому состоянию, в каком он был до команды make.

Запустите

make

Появились объектные файлы и исполняемый. Теперь

make clean

Объектные и исполняемый файлы исчезли. Остались только c-файлы, h-файл и сам Makefile. То есть, проект «очистился» от результатов команды make.

Правило install помещает исполняемый файл в каталог /usr/local/bin – стандартный каталог размещения пользовательских программ. Это значит, что её можно будет вызывать из любого места простым набором её имени.

Но помещать что-либо в этот каталог можно только, зайдя в систему под «суперпользователем». Для этого надо дать команду su и набрать пароль «суперпользователя». В противном случае система укажет, что вам отказано в доступе. Выход из «суперпользователя» осуществляется командой exit. Итак,

make

su

make install

exit

Теперь вы можете запустить это программу, просто введя имя программы, без прописывания пути.

kalkul

Можете открыть каталог /usr/local/bin. Там должен появиться файл с названием kalkul.

Давайте теперь «уберём за собой», не будем засорять систему.

su

make uninstall

exit

Посмотрите каталог /usr/local/bin. Файл kalkul исчез. Итак, правило uninstall удаляет программу из системного каталога.

Средства обеспечения переносимости и распространения

А теперь мы должны рассмотреть вопросы переносимости программ. Переносимость в мире UNIX имеет чрезвычайно большое значение. Количество реализаций UNIX, её клонов и UNIX-подобных систем весьма велико, и невозможно заранее предвидеть, на какой из них будет работать ваша программа. В разных реализациях могут быть разные версии компилятора, может быть по-разному организована система стандартных

каталогов для исполняемых файлов. Предлагать пользователю самостоятельно редактировать make-файлы в соответствии со своей системой – дело недопустимое. Необходимо, чтобы пользователь мог установить программу всего за несколько простых шагов. Следовательно, разработчик должен приложить к программе скрипт, который исследовал бы платформу пользователя и, в соответствии с ней, автоматически сгенерировал наиболее подходящий make-файл. Ясно, что такой скрипт будет очень сложным по структуре и написание его вручную чревато многочисленными ошибками. К счастью, этого делать и не надо. В наборе инструментов GNU имеются средства, автоматически создающие такой скрипт (иногда они упоминаются под собирательным названием Autotools).

Итак, пользователь получает ваш дистрибутив, состоящий из файлов исходного кода, среди этих файлов должен быть и тот самый вышеуказанный скрипт. Его общепринятое название – configure. Далее перед пользователем стоят всего две задачи. Первая – запустить этот скрипт.

./configure

В результате должен сгенерироваться Makefile. А, поскольку есть Makefile, то вторая задача – запустить хорошо знакомую нам программу make.

make

Будет собран запускаемый файл вашей программы. Пользователь может поместить его в стандартный каталог с исполняемыми программами с помощью команды make install (предварительно зайдя в систему в режиме суперпользователя), удалить его оттуда командой make uninstall, очистить дистрибутив от сгенерированных файлов командой make clean и т. д.

Итак, задача разработчика – подготовить скрипт configure.

Вначале создадим в каталоге проекта всего два простых файла: Makefile.am и configure.ac (желательно перед этим удалить оттуда все файлы, что мы создали раньше, оставить только problem.h, problem.cpp, main.cpp).

Makefile.am

bin_PROGRAMS=kalkul

kalkul_SOURCES=problem.h problem.cpp main.cpp

configure.ac

AC_INIT(main.cpp)

AM_INIT_AUTOMAKE(kalkul,0.1)

AC_PROG_CC

AC_PROG_CXX

AC_PROG_INSTALL

AC_OUTPUT(Makefile)

Этого вполне достаточно для формирования нужного нам скрипта. bin_PROGRAMS=kalkul указывает, как должен называться конечный запускаемый файл. kalkul_SOURCES=problem.h problem.cpp main.cpp указывает на все исходные файлы, участвующие в сборке этой программы. Если бы наша программа называлась как-нибудь по-другому (например, reader), то вместо kalkul_SOURCES стояло бы reader_SOURCES и т. д.

Скрипт configure.ac должен всегда начинаться директивой AC_INIT и заканчиваться AC_OUTPUT. Его команды означают следующее.

AC_INIT(main.cpp) является инициализацией этого скрипта. Ему в качестве параметра передаётся название любого из исходных файлов. Он проверяет, находится ли в данном каталоге такой файл. И, если он находится, значит, каталог действительно является рабочим.

AM_INIT_AUTOMAKE(kalkul,0.1) указывает, что мы будем использовать утилиту automake. Параметры указывают на название и версию, которые программа должна получить после сборки.

AC_PROG_CC и AC_PROG_CXX указывают, каким синтаксисом и с применением каких библиотек программа написана. Это помогает выбрать соответствующий компилятор. Наша программа написана целиком на C++ и

использует стандартную библиотеку C++, поэтому здесь мы можем оставить только AC_PROG_CXX. Если бы мы использовали предыдущую, C-версию программы, можно было бы оставить только AC_PROG_CC. В крупных программах, где используются элементы и того, и другого, пишутся обе директивы.

AC_PROG_INSTALL указывает, что надо в make-файле сформировать цель install, чтобы пользователь мог командой make install установить программу в системный каталог.

AC_OUTPUT завершает скрипт и указывает, что конечный файл должен называться Makefile.

Теперь в консоли заходим в каталог проекта, и даём по очереди следующие команды:

aclocal

autoconf

touch README AUTHORS NEWS ChangeLog

automake -a

./configure

make

После всего этого, если всё прошло нормально, в проекте должен появиться исполняемый файл. Теперь, если вы хотите сформировать дистрибутив для распространения, дайте команду

make dist

В папке проекта должен появиться архив kalkul-0.1.tar.gz. Можете теперь его выкладывать в интернете или рассылать по электронной почте.

Что же должен сделать получатель этого? Вначале распаковать архив:

gunzip kalkul-0.1.tar.gz

tar xf kalkul-0.1.tar

Затем зайти в каталог с дистрибутивом:

cd kalkul-0.1

Автоматически сформировать Makefile, который будет зависеть от конфигурации его операционной системы:

./configure

Затем собрать программу:

make

И установить её в системном каталоге, предварительно зайдя в режиме суперпользователя:

su

make install

exit

Чтобы удалить её, нужно указать make uninstall в режиме суперпользователя:

su

make uninstall

exit

Разберём команды, которые мы сейчас вводили:

aclocal сканирует файл configure.ac и, в зависимости от его директив, формирует макросы, предназначенные для autoconf и automake; эти макросы сохраняются в файле aclocal.m4;

autoconf формирует скрипт configure в зависимости от содержимого configure.ac;

automake формирует скрипт Makefile.in в зависимости от содержимого Makefile.am; в дальнейшем пользователь, запустив configure, сформирует Makefile на основании Makefile.in. Флаг -a означает, что, если программа не найдёт в каталоге проекта файлов install-sh, missing, INSTALL, COPYING, depcomp, она автоматически создаёт их.

Если мы просто запустим команды в таком порядке, то automake выдаст много «жалоб» и откажется работать. Дело в том, что дополнительная задача

automake – проверка соответствия нашего дистрибутива стандартам GNU. В соответствии с этими стандартами, в дистрибутиве обязательно должны присутствовать файлы NEWS, README, AUTHORS, ChangeLog. Создадим эти файлы командой touch NEWS README AUTHORS ChangeLog. Системная утилита touch меняет время модификации файла на текущее. Если указанного файла нет, то он создаётся пустым.

Вообще, если уж мы начали разговор о стандартах GNU, нам придётся признать, что мы сейчас сделали всё неправильно. Мы сильно упростили задачу, чтобы сделать её более наглядной.

По стандарту в корневом каталоге проекта расположены только текстовые файлы README, INSTALL, AUTHORS, THANKS, ChangeLog, COPYING, а также файлы, имеющие отношение к сборке программы. Все основные файлы проекта обычно располагаются во вложенных каталогах. Вложенные каталоги обычно следующие:

src – содержит все файлы исходного кода;

lib – необязательный каталог; содержит исходные коды библиотек, используемых сразу несколькими программами, а также участков кода, имеющих отношение к переносимости программ;

doc – содержит документацию к программе;

m4 – содержит макросы для autoconf, которые вы, возможно, захотите передать другим разработчикам;

intl – содержит участки программы, отвечающие за интернационализацию программы, то есть перевода её интерфейса на разные национальные языки;

po – содержит списки сообщений, с помощью которых программа общается с пользователем; здесь же – варианты этих сообщений на языках разных народов, предназначенные для интернационализации.

Возможны и другие варианты.

Давайте создадим в нашем проекте каталог src и перенесём туда файлы problem.h, problem.cpp, main.cpp. Все остальные файлы и каталоги удалим.

В корневом каталоге проекта создадим файл Makefile.am. Поместим в него следующий текст.

```
SUBDIRS = src
```

В каталоге src создадим другой файл с точно таким же названием Makefile.am. Его содержимое:

```
bin_PROGRAMS=kalkul
kalkul_SOURCES=problem.h problem.cpp main.cpp
```

В корневом каталоге создадим файл configure.ac. Текст его следующий.

```
AC_INIT(src/main.cpp)
AM_INIT_AUTOMAKE(kalkul,0.1)
AC_PROG_CC
AC_PROG_CXX
AC_PROG_INSTALL
AC_OUTPUT(Makefile src/Makefile)
```

Обратите внимание: в начальном макросе AC_INIT мы указываем, что исходные файлы расположены в каталоге src, а в конечном AC_OUTPUT указываем сразу на оба формируемых файла Makefile.

Сгенерируем конфигурационный скрипт обычным образом.

```
aclocal
autoconf
touch README AUTHORS NEWS ChangeLog
automake -a
```

Соберём программу.

```
./configure
make
```

В папке src появился исполняемый файл. Теперь создадим архив.

make dist

Обратите внимание на такую строку в сгенерированном make-файле:

DEFS = -DPACKAGE_NAME=\"\" -DPACKAGE_TARNAME=\"\" -DPACKAGE_VERSION=\"\" -DPACKAGE_STRING=\"\" -DPACKAGE_BUGREPORT=\"\" -DPACKAGE=\"kalkul\" -DVERSION=\"0.1\"

Таким путём make-файл указывает на имя и версию нашего пакета. Мы передали эти параметры с помощью директивы AM_INIT_AUTOMAKE(kalkul, 0.1) в файле configure.ac. Мы можем таким же образом передать и много других параметров. Если их будет слишком много, то эта строка в make-файле станет слишком длинной. Это чревато следующими трудностями. Во-первых, в такой длинной строке трудно визуально найти ошибки. Во-вторых, некоторые UNIX-системы имеют лимит на длину строки в скриптах, и, если мы превысим этот лимит, скрипт не будет обрабатываться.

Существует и другой способ, при котором эти параметры указываются в заголовочном файле config.h с помощью макроса #define. Давайте перепишем сборочные скрипты с использованием этого заголовочного файла. Сам файл будет сформирован автоматически. Нам же нужно внести небольшие изменения в скрипты.

Удалите из проекта все файлы, кроме исходников (problem.h, problem.cpp, main.cpp).

В начало каждого из этих файлов исходников необходимо вставить макрос, требующий включения в них заголовочного файла config.h (и добавить к нему макрос защиты от повторного включения того же файла). Включите следующий текст в начало файлов main.cpp, problem.cpp, problem.h

```
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif
```

В корневом каталоге проекта создайте новый configure.ac.

AC_INIT(src/main.cpp)

AM_CONFIG_HEADER(src/config.h)

AM_INIT_AUTOMAKE(kalkul,0.1)

AC_PROG_CC

AC_PROG_CXX

AC_PROG_INSTALL

AC_OUTPUT(Makefile src/Makefile)

От предыдущего он отличается только тем, что содержит директиву AM_CONFIG_HEADER(src/config.h), указывающую, что такой заголовочный файл должен быть сформирован. Оба файла Makefile.am остаются без изменений.

В корневом каталоге проекта:

SUBDIRS = src

В каталоге src:

bin_PROGRAMS=kalkul

kalkul_SOURCES=problem.h problem.cpp main.cpp

Команды по формированию конфигурационного скрипта те же, за исключением того, что к ним добавляется autoheader. Обратите внимание: эти команды должны даваться строго в порядке, указанном ниже:

aclocal

autoconf

touch NEWS README AUTHORS ChangeLog

autoheader

automake -a

В результате, помимо всего прочего, сформировался файл config.h.in, в котором указаны все макросы препроцессора, которые войдут в будущий заголовочный файл config.h. Сам же config.h будет формироваться на машине

пользователя по команде ./configure, и его структура будет зависеть от конфигурации компьютера пользователя. Давайте проделаем работу пользователя.

./configure

make

Посмотрим теперь на Makefile. Вместо длинной строки, указанной выше, мы видим компактную запись, указывающую, что необходимую информацию можно найти в заголовочном файле.

DEFS = -DHAVE_CONFIG_H

Теперь, как обычно, создадим дистрибутив:

make dist

Давайте теперь всю эту же процедуру проделаем с предыдущей версией нашего калькулятора, написанной на языке C. Если вы помните, там мы вынуждены были явным образом подключать математическую библиотеку libm, потому что компилятор C неявно её не подключает. Собственно, здесь нам всего лишь нужно добавить одну строчку. Но рассмотрим всё по порядку. Создадим новый каталог для проекта kalkulc. Внутри него создадим вложенный каталог src, и перенесём в каталог src все три файла от C-версии нашего калькулятора. Это main.c, calculate.c, calculate.h. Только обратите внимание, что в прошлый раз, работая с этой версией калькулятора, мы не пользовались заголовочным конфигурационным файлом config.h, поскольку мы тогда не пользовались инструментами автогенерации make-файла. Теперь же мы будем пользоваться всеми полученными знаниями в полном объёме, так что этот заголовочный файл нам понадобится.

Вставьте в самое начало каждого из этих трёх файлов (main.c, calculate.c и calculate.h) макрос, требующий включения этого файла. И, заодно, защиту от его повторного включения.

#ifdef HAVE_CONFIG_H

#include <config.h>

#endif

В каталоге проекта создадим файл configure.ac со следующим текстом.

AC_INIT(src/main.c)

AM_CONFIG_HEADER(src/config.h)

AM_INIT_AUTOMAKE(kalkul,0.1)

AC_PROG_CC

AC_PROG_CXX

AC_PROG_INSTALL

AC_OUTPUT(Makefile src/Makefile)

Как видите, он аналогичен тому, что мы делали в C++-версии.

Здесь же создаём файл Makefile.am с одной строчкой.

SUBDIRS = src

А внутри каталога src – ещё один Makefile.am.

bin_PROGRAMS = kalkul

kalkul_SOURCES = calculate.h calculate.c main.c

kalkul_LDADD = -lm

Как видите, что он отличается от C++-версии только строчкой kalkul_LDADD = -lm Эта строчка и указывает, какую библиотеку следует подключить при компиляции.

Все остальные шаги вам уже хорошо знакомы.

aclocal

autoconf

touch NEWS README AUTHORS ChangeLog

autoheader

automake -a

./configure

make

Задание на выполнение.

1. Скомпилировать простое приложение с помощью gcc
2. Скомпилировать приложение из нескольких исходных файлов, с отдельной фазой компоновки.
3. Создать makefile для автоматической сборки приложения.
4. Собрать zlib из исходных текстов.

Практическое занятие № 5. Работа с Elecard STB SDK.

Цель работы.
Познакомиться с кросс-компиляцией на примере ELDK.

Задание на выполнение.

1. Установить Elecard STB SDK в соответствии с инструкцией.
2. Настроить ЦТП на загрузку по NFS. Научиться грузить ЦТП с нужного NFS сервера вручную через меню загрузчика и с помощью специальной прошивки.
3. Портировать свое приложение на MIPS, выполнить его на ЦТП.
4. Портировать zlib на ЦТП.

Практическое занятие № 6. Разработка и отладка программного обеспечения для PNX8950, система сборки, загрузка и отладка на устройстве ЦТП 820.

Цель работы.
Ознакомиться с технологией создания прошивок для ЦСП PNX8950, настройка загрузки ЦТП 820 по nfs.

Задание на выполнение.

1. Ознакомление с системой сборки.
2. Создание образа ЦСП, загрузка и старт по nfs с рабочего компьютера.
3. Изучение возможных кодов ошибок и решение проблем.

Практическое занятие № 7. Разработка и отладка программного обеспечения для PNX8950, создание программы «Hello world»

Цель работы.
Разобраться в примере программы. Изучить команды и параметры makefile-а проекта. Научится создавать программы, которые запускаются на ЦТП 820.

Задание на выполнение.

1. Создание программы «Hello world!».
2. Разобраться в интерфейсе Elecard-овских компонент для декодирования аудио.
3. Создать программу декодирования аудио потока MP3/AAC/Vorbis (по выбору) на основе изученных компонент.

Практическое занятие № 8. Разработка и отладка программного обеспечения для PNX8950, программы состоящей из нескольких частей, одна работающая на MIPS-е другая на TriMedia.

Цель работы.

Ознакомиться с технологией создания мультимедийных приложений NDK фирмы NXP-Trident и получить навыки разработки TSSA модулей. Изучение примера программы, работающей на двух ядрах PNX8950 MIPS и TriMedia.

Задание на выполнение.

1. Создать TSSA модуль.
2. Создать приложение, использующее данный модуль.
3. Организовать пересылку команд между созданными модулями.

Практическое занятие № 9. Разработка и отладка программного обеспечения для PNX8950, использование ассемблера.

Цель работы.

Получить навыки профилирования и усовершенствования TSSA модулей. Создание специальных вставок на ассемблере TriMedia.

Задание на выполнение.

1. Разработать ассемблерную функцию. Отладить ее на симуляторе и на устройстве.

2. Провести профилирование TSSA модуля. Найти его «тонкое» место. Выделить его в отдельную функцию. Запустить функцию на симуляторе. Ускорить ее написание ассемблера.
3. Провести профилирование результата на устройстве. Сделать выводы.

Практическое занятие № 10. Сборка модулей (драйверов) ядра. Загрузка/выгрузка модулей ядра. Просмотр системных сообщений сообщения уровня ядра.

Цель работы.

Получение навыков работы с модулями (драйверами) ядра операционной системы Linux. Получение навыков обработки сообщений ядра.

Задание на выполнение.

1. Распаковка исходников ядра Linux. Обзор make системы сборки ядра. Создание makefil-а для модуля ядра. Сборка модуля и его установка в рабочую систему.
2. Просмотр информации о модуле (modinfo). Получение описания модулей, версии ядра и прочей информации. Загрузка модуля ядра с помощью утилиты insmod с учетом зависимостей. Прослеживание сообщений от загружаемого модуля ядра. Выгрузка модуля и его зависимостей.
3. Загрузка модуля ядра с помощью утилиты modprobe. Отслеживание загруженных модулей с помощью утилиты lsmod. Прослеживание сообщений от загружаемого модуля и его зависимостей.
4. Добавление вывода printk уровня ядра в модуль. Отслеживание добавленных сообщений уровня ядра в системном логе. Использование предопределенных макросов ядра LINUX_VERSION_CODE, KERNEL_VERSION и KERN_INFO.

Практическое занятие № 11. Реализация драйвера с обменом данными между пространствами ядра и приложений.

Цель работы.

Получение и закрепление навыков разработки драйверов ядра, обмена данными с пользовательскими приложениями.

Задание на выполнение.

Реализовать драйвер с созданием файла в procfs. Драйвер должен обрабатывать чтение и запись файла.

1. Драйвер должен: хранить до 8-ми значений типа char, которые передаются при попытке записи, а при чтении файла драйвера возвращать запомненные значения.

Например: файл драйвера - /proc/buffer

user@mach~$ echo "H" > /proc/buffer

user@mach~$ echo "e" > /proc/buffer

user@mach~$ echo "l" > /proc/buffer

user@mach~$ echo "l" > /proc/buffer

user@mach~$ echo "o" > /proc/buffer

user@mach~$ echo "!" > /proc/buffer

user@mach~$ cat /proc/buffer

Hello!

user@mach~$ echo "1" > /proc/buffer

user@mach~$ echo "2" > /proc/buffer

user@mach~$ echo "3" > /proc/buffer

user@mach~$ echo "4" > /proc/buffer

user@mach~$ echo "5" > /proc/buffer

user@mach~$ echo "6" > /proc/buffer

user@mach~$ echo "7" > /proc/buffer

user@mach~$ echo "8" > /proc/buffer

user@mach~$ echo "9" > /proc/buffer

user@mach~$ cat /proc/buffer

23456789

2. Драйвер должен выполнять арифметические действия: сложение, вычитание и умножение. При записи в файл драйвера передается строка вида

<операнд1> <оператор> <операнд2>, где <операнд1> и <операнд2> - положительные числа в десятичном представлении в пределах [0..99999999], <оператор> - '+', '-' или '*'. При чтении файла драйвера должен возвращаться результат операции.

Пример: файл драйвера - /proc/calc

user@mach~$ echo "2 * 2" > /proc/calc

user@mach~$ cat /proc/calc

4

user@mach~$ echo "12345679 * 8" > /proc/calc

user@mach~$ cat /proc/calc

98765432

3. Драйвер должен выполнять побитовые логические операции: | (или) и & (и). При записи в файл драйвера передается строка вида <операнд1> <оператор> <операнд2>, где <операнд1> и <операнд2> - положительные числа в шестнадцатеричном представлении (int - 4 байта, с обязательным префиксом 0x), <оператор> - '|' или '&'. При чтении файла драйвера должен возвращаться результат операции.

Пример: файл драйвера - /proc/logical

user@mach~$ echo "0x23 | 0x56" > /proc/logical

user@mach~$ cat /proc/logical

0x77

user@mach~$ echo "0x78 & 0xC5" > /proc/logical

user@mach~$ cat /proc/calc

0x40

Практическое занятие № 12. Создание нод устройств. Работа с нодами в командной строке. Разбор драйвера символьного устройства. Разбор программы с системным вызовом ioctl.

Цель работы.

Закрепление использования bash-скриптов. Получение навыков работы с фалами устройств. Создание символьных и блочных файлов устройств.

Получение навыков регистрация символьного устройства в ядре Linux, описания функций чтения, записи и функции системного вызова ioctl.

Задание на выполнение.

1. Реализовать скрипт обзора файловой системы sysfs и поиска и создания нод SATA устройств (разделов жестких дисков) в домашней директории. Скрипт должен создавать устройства дисков с правильными majornum и minornum.
2. Реализовать скрипт разбора зарегистрированных символьных и блочных устройств в системе (/proc/devices), а также создания файла устройства, определенного по имени.
3. Написание драйвера символьного устройства «Hello world». С фиксированным majornum=200. При чтении файла устройства передавать в буфер чтения приветственное сообщение. При записи в файл устройства выводить переданное сообщение в лог ядра.

Пример:

```
user@mach~$ cat /dev/hello
1st hello!!
user@mach~$ cat /dev/hello
2nd hello!!
user@mach~$ echo «Hi» > /dev/hello
user@mach~$ tail -1 /var/log/messages
Jun  1 14:03:25 elecard-serga1sample[1234]: Hi
```

4. Написание драйвера символьного устройства с реализацией системного вызова ioctl. С динамическим выделением majornum. Написание тестового приложения реализующее вызов ioctl этого устройства.

Практическое занятие № 13. Кросс-компиляция модулей ядра. Разбор примера работы с портами ввода/вывода для DaVinci.

Цель работы.

Закрепление навыков кросс-компиляции.

Получение навыков компиляции модулей ядра под кросс-системы и установки модулей в целевую файловую систему.

Получение навыков работы с регистрами (физической памятью) и портами.

Навыки работы с портами ввода/вывода общего назначения.

Задание на выполнение.

1. Кросс-компилирование модуля ядра реализующее символьное устройство «Hello world». Запуск на платформе DaVinci и отладка.
2. Кросс-компилирование модуля ядра, реализующего обмен данными между пространством приложений и пространством ядра при помощи файловой системы procfs. Запуск на платформе DaVinci и отладка.
3. Изучение блока портов ввода/вывода общего назначения (GPIO) процессора TI DaVinic. Реализация драйвера конфигурации произвольного GPIO. Задание параметров GPIO осуществляется через запись в файл устройства. Формат записи: conf <номер банка> <номер пина> <вход=0/выход=1> - конфигурация GPIO

set <номер банка> <номер пина> <выходное значение 0,1> - задание выхода

net <номер банка> <номер пина> - чтение входа

Например:

user@mach~$ echo «conf 2 5 0» > /dev/sample_gpio

```
user@mach~$ echo «get 2 5» > /dev/sample_gpio
0
```

Практическое занятие № 14. Симуляция XDAIS модуля.

Цель работы.

Создать модуль программного обеспечения XDAIS. Получить и закрепить навыки создания межпроцессорного взаимодействия на базе микропроцессора DaVinci с использованием технологии DspLink. Ознакомиться с системой TI CCS. Получить навыки симуляции и профилирования программных модулей для чипов фирмы TI. Получить навыки запуска и профилирования XDAIS модулей в системе CCS.

Задание на выполнение.

1. Разработка XDAIS модуля по шаблону.
2. Создание программного обеспечения, которое связывает процессор общего назначения и ЦСП.
3. Выполнение отладки XDAIS модуля.
4. Создать приложение в системе CCS.
5. Провести симуляцию данной задачи.
6. Провести профилирование задачи средствами CCS.
7. Провести сборку XDAIS модуля в системе CCS.
8. Создать приложение для запуска модуля.
9. Провести симуляцию и профилирование модуля.
10. Сравнить результаты с результатами профилирования на устройстве.

Практическое занятие № 15. Оптимизация С кода и использование директив компилятора.

Цель работы.

Получить навыки оптимизации С кода для ЦСП. Оценить степень важности правильной работы с памятью.

Задание на выполнение.

1. Оптимизировать С код компоненты согласно методам, приведенным на лекционных занятиях.
2. Провести профилирование компоненты на устройстве и симуляторе CCS.

Практическое занятие № 16. Структура ассемблера.

Цель работы.

Получить навыки создания ассемблерного кода для микропроцессоров С64х.

Задание на выполнение.

1. Создать реализацию «тонкого» места компоненты на ассемблере.
2. Провести профилирование.
3. Сделать выводы

Практическое занятие № 17. Оптимизация модуля фильтра БИХ с использованием ассемблера.

Цель работы.

Получение и закрепление навыков параллельного программирования на ЦСП С64х. Использование конвейерных циклов.

Задание на выполнение.

1. Провести расчет минимального шага цикла задачи с использованием инструкций процессора.

2. Провести разработку предварительной части цикла, тела цикла и окончательной стадии цикла.
3. Провести усовершенствование функции согласно полученным данным

Практическое занятие № 18. Оптимизация модуля фильтра КИХ с использованием ассемблера.

Цель работы.

Получение и закрепление навыков параллельного программирования на ЦСП С64х.

Задание на выполнение.

1. Провести расчет минимального шага цикла задачи с использованием инструкций процессора.
2. Провести разработку предварительной части цикла, тела цикла и окончательной стадии цикла.
3. Провести усовершенствование функции согласно полученным данным.