

Утверждаю

Ректор университета

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_О.Н.Федонин

«\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_2020 г.

**UNIX-СИСТЕМЫ**

**ПРОГРАММИРОВАНИЕ ПОД UNIX НА ЯЗЫКИ C**

Методические указания

к выполнению лабораторной работы №5

для студентов очной формы обучения

по направлениям подготовки

09.03.01 «Информатика и вычислительная техника»,

09.03.04 «Программная инженерия»,

02.03.03 «Математическое обеспечение и администрирование информационных систем»

**Брянск 2020**

**УКД 004.43**

UNIX-системы. Программирование под UNIX на языке C [Электронный ресурс]: методические указания к выполнению лабораторной работы №5 для студентов очной формы обучения по направлениям подготовки 09.03.01 «Информатика и вычислительная техника», 09.03.04 «Программная инженерия», 02.03.03 «Математическое обеспечение и администрирование информационных систем». – Брянск: БГТУ, 2020. – 31 с.

Разработал

Е.О.Трубаков,

канд. техн. наук, доц.

Рекомендовано кафедрой «Информатика и программное обеспечение» БГТУ (протокол №5 от 26.01.2020)

**Методические указания публикуются в авторской редакции**

# Цель работы

Цель работы – приобретение навыков программирования в ОС Unix, таких как: ознакомиться с компиляторами языка C, используемыми в ОС Unix; изучить технологию автоматизации сборки программ в ОС Unix (утилита make); ознакомится с примерами программирования на базе исходных текстов стандартных утилит Unix; получить практические навыки разработки программ для ОС Unix.

Продолжительность работы – 2 часа.

# Теоретическая часть

Трудно переоценить значимость языка программирования C в области компьютерного программного обеспечения. На этом языке написана существенная часть кода современных операционных систем и приложений. Преобладающее число современных языков прямо или косвенно унаследовало от языка C синтаксис или важнейшие концепции программирования. Даже компиляторы и интерпретаторы многих из этих языков написаны на языке C. Уже три спецификации языка С в разные годы опубликованы в качестве международных стандартов . Однако, не так широко известно, как тесно связана история появления и развития языка C операционной системой Unix.

Язык С был разработан в начале 1970-х годов Кеном Томпсоном и Денисом Ритчи для использования в операционной системе Unix. Появление этого языка позволило сделать код Unix платформенно-переносимым – достаточно было переписать код компилятора для новой архитектуры ЭВМ и можно было запускать операционную систему и весь набор приложений для нее на новой аппаратной платформе. Код современных версий Unix-систем практически целиком написан на языке C. И несмотря на то что с приложениями для ОС Unix дело обстоит другим образом – они написаны на различных языках программирования, язык С остается традиционным и по сути стандартным языком программирования для ОС Unix. Так, большинство актуальных стандартов Unix неразрывно связаны с языком C – фактически прикладной программный интерфейс всех подсистем ОС Unix описан на этом языке. Нередко стандартные инсталляции многих версий ОС Unix включают в себя компилятор языка C.

Пример программы, осуществляющий передачу символов из стандартного ввода на стандартный вывод представлен:

|  |
| --- |
| #include <stdio.h>  int main() {  char c;  while((c = getchar()) != EOF)  {  putchar(c);  }  return 0;  } |

Пример программы, выполняющей обработку аргументов, приведен:

|  |
| --- |
| #include <ctype.h>  #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  int main (int argc, char \*\*argv) {  int index;  int c;  opterr = 0;  while ((c = getopt (argc, argv, "abc:")) != -1)  switch (c)  {  case 'a':  printf("Опция '-a'\n");  break;  case 'b':  printf("Опция '-b'\n");  break;  case 'c':  printf("Опция '-c' с аргументом '%s'\n", optarg);  break;  case '?':  if(optopt == 'c')  fprintf(stderr,"Для опции '-%c' требуется аргумент.\n", optopt);  else if (isprint(optopt))  fprintf (stderr, "Неизвестная опция '%c'.\n", optopt);  else fprintf (stderr, "Неизвестный символ '\\x%x'.\n", optopt);  return 1;  default:  abort();  }  for (int index = optind; index < argc; index++)  printf ("Параметр %s\n", argv[index]);  return 0;  } |

## Сведения о Makefile

Допустим, вы разрабатываете некую программу под названием foo, состоящую из пяти заголовочных файлов – 1.h, 2.h, 3.h, 4.h и – 5.h, и шести файлов с исходным текстом программы на языке С – 1.cpp, 2.cpp, 3.cpp, 4.cpp, 5.cpp и main.cpp (хочу заметить, что в реальных проектах следует избегать подобного стиля именования файлов).

Теперь представим себе, что вы обнаружили ошибку в файле 2.cpp и исправили ее. Далее, чтобы получить исправленную версию программы вы компилируете все файлы, входящие в состав проекта, хотя изменения коснулись только одного файла. Это приводит к нерациональной потере времени, особенно если компьютер не слишком быстрый.

Решение проблемы - утилита make.

Вместо того чтобы производить повторную компиляцию всех файлов с исходными текстами, она обрабатывает только те файлы, которые претерпели изменения. В нашем случае будет скомпилирован только один файл – 2.cpp.

Кроме того:

* утилита *make* значительно упрощает жизнь, когда для сборки проекта необходимо выполнение длинных и сложных команд;
* проект иногда требует задания редко используемых, а потому сложных для запоминания опций компилятора. *make* избавит вас от необходимости удерживать их в памяти;
* единообразие, т.к. работа с этой утилитой поддерживается многими средами разработки;
* процесс сборки можно автоматизировать, поскольку *make* может быть вызвана из сценариев или из *cron*.

## Для чего нужен Makefile

Несмотря на все свои достоинства, утилита *make* ничего не знает о нашем проекте, поэтому необходимо создать простой текстовый файл, который будет содержать все необходимые инструкции по сборке. Файл с инструкциями по сборке проекта называется *makefile*.

Как правило, этим файлам дается имя *makefile* или *Makefile*, в соответствии с соглашениями по именованию таких файлов. Если же вы дадите файлу инструкций другое имя, то вам потребуется вызывать утилиту make с ключом –f.

Например, если свой *makefile* вы назвали *my\_make*, то команда на сборку проекта будет выглядеть так:

|  |
| --- |
| make -f my\_make |

## Структура файла

*Makefile* содержит разделы для "целей" *[targets]*, зависимостей *[dependencies]* и правил *[rules]* сборки. Все это оформляется следующим образом: сначала указывается имя цели (обычно это имя исполняемого или объектного файла), после которого следует двоеточие, затем следуют имена зависимостей, т.е. файлов, необходимых для получения данной цели. И, наконец, следует список правил: т.е. команд, которые необходимо выполнить для получения указанной цели.

Простой пример структуры *makefile*:

|  |
| --- |
| target: dependencies  command  command  ... |

Каждое правило *command* должно начинаться с символа табуляции – это обязательное условие. Отсутствие символа табуляции в начале строки с правилом – самая распространенная ошибка. Но подобные ошибки легко обнаруживаются, так как утилита *make* сообщает о них.

## Пример Makefile

Ниже приводится простой пример (номера строк добавлены для ясности).

|  |
| --- |
| 1 client: conn.o  2 g++ client.cpp conn.o -o client  3 conn.o: conn.cpp conn.h  4 g++ -c conn.cpp -o conn.o |

В этом примере строка, содержащая текст client: conn.o, называется "строкой зависимостей", а строка g++ client.cpp conn.o -o client называется "правилом" и описывает действие, которое необходимо выполнить.

А теперь более подробно о примере, приведенном выше:

* задается цель – исполняемый файл client, который зависит от объектного файла *conn.o*;
* правило для сборки данной цели;
* в третьей строке задается цель *conn.o* и файлы, от которых она зависит – *conn.cpp* и *conn.h*;
* в четвертой строке описывается действие по сборке цели *conn.o*.

## Комментарии

Строки, начинающиеся с символа "#", являются комментариями. Ниже приводится пример makefile с комментариями:

|  |
| --- |
| 1 # Создать исполняемый файл "client"  2 client: conn.o  3 g++ client.cpp conn.o -o client  4  5 # Создать объектный файл "conn.o"  6 conn.o: conn.cpp conn.h  7 g++ -c conn.cpp -o conn.o |

## «Ложная» цель

Обычно «ложные» *[phony]* цели, представляющие «мнимое» имя целевого файла, используются в случае возникновения конфликтов между именами целей и именами файлов при явном задании имени цели в командной строке.

Допустим в makefile имеется правило, которое не создает ничего, например:

|  |
| --- |
| clean:  rm \*.o temp |

Поскольку команда rm не создает файл с именем *clean*, то такого файла никогда не будет существовать и поэтому команда *make clean* всегда будет отрабатывать.

Однако данное правило не будет работать, если в текущем каталоге будет существовать файл с именем *clean*. Поскольку цель *clean* не имеет зависимостей, то она никогда не будет считаться устаревшей и, соответственно, команда «rm \*.o temp» никогда не будет выполнена (при запуске make проверяет даты модификации целевого файла и тех файлов, от которых он зависит. И если цель оказывается "старше", то make выполняет соответствующие команды-правила). Для устранения подобных проблем и предназначена специальная декларация .PHONY, объявляющая «ложную» цель. Например:

|  |
| --- |
| .PHONY: clean |

Таким образом, мы указываем необходимость исполнения цели, при явном ее указании, в виде make clean вне зависимости от того – существует файл с таким именем или нет.

## Переменные

Определить переменную в *makefile* можно следующим образом:

|  |
| --- |
| . $VAR\_NAME=value |

В соответствии с соглашениями имена переменных задаются в верхнем регистре:

|  |
| --- |
| $OBJECTS=main.o test.o |

Чтобы получить значение переменной, необходимо ее имя заключить в круглые скобки и перед ними поставить символ '$', например:

|  |
| --- |
| $(VAR\_NAME) |

В makefile-ах существует два типа переменных: «упрощенно вычисляемые» и «рекурсивно вычисляемые». В рекурсивно вычисляемых переменных все ссылки на другие переменные будут замещены их значениями, например:

|  |
| --- |
| TOPDIR=/home/tedi/project  SRCDIR=$(TOPDIR)/src |

При обращении к переменной SRCDIR вы получите значение /home/tedi/project/src.

Однако рекурсивные переменные могут быть вычислены не всегда, например, следующие определения:

|  |
| --- |
| CC = gcc -o  CC = $(CC) -O2 |

Они выльются в бесконечный цикл. Для разрешения этой проблемы следует использовать «упрощенно вычисляемые» переменные:

|  |
| --- |
| CC := gcc -o  CC += $(CC) -O2 |

Где символ ':=' создает переменную CC и присваивает ей значение «gcc –o». А символ '+=' добавляет «-O2» к значению переменной CC.

## Библиотеки

Библиотека объектных файлов – это файл, содержащий несколько объектных файлов, которые будут использоваться вместе в стадии линковки программы. Нормальная библиотека содержит символьный индекс, который состоит из названий функций и переменных и т.д., которые содержатся в библиотеке. Это позволяет ускорить процесс линковки программы, так как поиск функций и переменных в объектных файлах библиотеки происходит намного быстрее, чем поиск в наборе указанных объектных файлов. Поэтому использование библиотеки позволяет компактно хранить все требуемые объектные файлы в одном месте, и при этом значительно повысить скорость компиляции.

Объектные библиотеки по способу использования разделяются на два вида:

* статические библиотеки;
* динамические библиотеки.

**Статическая библиотека** – это коллекция объектных файлов, которые присоединяются к программе во время линковки программы. Таким образом статические библиотеки используются только при создании программы. Потом в работе самой программы они не принимают участие, в отличие от динамических библиотек.

**Динамическая библиотека** – это созданная специальным образом библиотека, которая присоединяется к результирующей программе в два этапа. Первый этап, это, естественно, этап компиляции. На этом этапе линковщик встраивает в программу описания требуемых функций и переменных, которые присутствуют в библиотеке. Сами объектные файлы из библиотеки не присоединяются к программе. Присоединение этих объектных файлов (кодов функций) осуществляет системный динамический загрузчик во время запуска программы. Загрузчик проверяет все библиотеки, прилинкованные к программе, на наличие требуемых объектных файлов, затем загружает их в память и присоединяет их в копии запущенной программы, находящейся в памяти.

Сложный процесс загрузки динамических библиотек замедляет запуск программы, но у него есть существенный, даже можно сказать неоценимый плюс – если другая запускаемая программа линкована с этой же загруженной динамической библиотекой, то она использует ту же копию библиотеки. Это означает, что требуется гораздо меньше памяти для запуска нескольких программ, сами загрузочные файлы меньше по размеру, что экономит место на дисках.

## Создание и использование статических библиотек

Пусть у нас имеется следующая программа:

|  |
| --- |
| /\* main.c\*/  #include <stdio.h>  // описываем функцию f1() как внешнюю  extern int f1();  // описываем функцию f2() как внешнюю  extern int f2();  int main()  {  int n1, n2;  n1 = f1();  n2 = f2();  printf("f1() = %d\n",n1);  printf("f2() = %d\n",n2);  return 0;  } |

Теперь создаем два файла, каждый из которых будет содержать полное определение внешней функции из главной программы (Листинг 4). Файлы назовем f1.c и f2.c:

|  |
| --- |
| /\* файл f1.c\*/  int f1()  {  return 2;  }  /\* файл f2.c \*/  int f2()  {  return 10;  } |

Для создания статических библиотек существует специальная простая программа, называемая ar (сокр. от *archiver* - архиватор). Она используется для создания, модификации и просмотра объектных файлов в статических библиотеках, которые в действительности представляют из себя простые архивы.

Создадим из f1.c и f2.c объектные файлы:

|  |
| --- |
| nyquist:~# gcc -c f1.c f2.c |

В результате получим два файла – f1.o и f2.o. Для того, чтобы создать библиотеку из объектных файлов надо вызвать программу ar со следующими параметрами:

|  |
| --- |
| ar rc libимя\_библиотеки.a [список\_\*.o\_файлов] |

Допустим наша библиотека будет называться fs, тогда команда запишется в виде:

|  |
| --- |
| nyquist:~# ar rc libfs.a f1.o f2.o |

В результате получим файл libfs.a, в котором будут лежать копии объектных файлов f1.o и f2.o. Если файл библиотеки уже существует, то архиватор будет анализировать содержимое архива, он добавит новые объектные файлы и заменит старые обновленными версиями. Опция c заставляет создавать (от create) библиотеку, если ее нет, а опция r (от replace) заменяет старые объектные файлы новыми версиями.

Пока у нас есть лишь архивный файл libfs.a. Чтобы из него сделать полноценную библиотеку объектных файлов надо добавить к этому архиву индекс символов, т.е. список вложенных в библиотеку функций и переменных, чтобы линковка происходила быстрее. Делается это командой:

|  |
| --- |
| ranlib libимя\_библиотеки.a |

Программа *ranlib* добавит индекс к архиву и получится полноценная статическая библиотека объектных файлов. Стоит отметить, что на некоторых системах программа ar автоматически создает индекс, и использование ranlib не имеет никакого эффекта. Но тут надо быть осторожным, при автоматической компиляции библиотеки с помощью файлов makefile, если вы не будете использовать утилиту ranlib, то возможно на каких-то системах библиотеки будут создаваться не верно и потеряется независимость от платформы. Так что возьмем за правило тот факт, что утилиту ranlib надо запускать в любом случае, даже если он нее нет никакого эффекта.

Для компиляции нашего основного файла main.c надо сообщить компилятору, что надо использовать библиотеки. Для того, чтобы компилятор знал, где искать библиотеки ему надо сообщить каталог, в котором они содержатся и список этих библиотек. Каталог с библиотеками указывается ключом -L, в нашем случае библиотека находится в текущем каталоге, значит, путь до нее будет в виде точки (-L.). Используемые библиотеки перечисляются через ключ -l, после которого указывается название библиотеки без префикса lib и окончания .a. В нашем случае этот ключ будет выглядеть, как -lfs. Теперь все одной командой:

|  |
| --- |
| nyquist:~# gcc -c main.c  nyquist:~# gcc main.o -L. -lfs -o rezult |

Или иначе:

|  |
| --- |
| nyquist:~# gcc main.c -L. -lfs -o rezult |

Заметьте, что компилятору нужны библиотеки на этапе создания конечного файла, т.е. линковки. В первом случае процесс компиляции совершается первой командой, а сборка файла второй командой. Если же мы попытаемся подсунуть библиотеку на этапе компиляции, то получим вежливый ответ:

|  |
| --- |
| nyquist:~# gcc -c main.c -L. -lfs  gcc: -lfs: linker input file unused since linking not done |

Что означает, что файлы библиотек не нужны, до процесса линковки. Данная команда создаст лишь файл *main.o*, который в итоге потом придется собирать отдельно.

## Создание и использование динамических библиотек

Обычный объектный файл, создаваемый приведенным ранее способом, вовсе не подходит для динамических библиотек. Связано это с тем, что все объектные файлы, создаваемые обычным образом, не имеют представления о том в какие адреса памяти будет загружена использующая их программа. Несколько различных программ могут использовать одну библиотеку, и каждая из них располагается в различном адресном пространстве. Поэтому требуется, чтобы переходы в функциях библиотеки (команды jmp на ассемблере) использовали не абсолютную адресацию, а относительную. То есть генерируемый компилятором код должен быть независимым от адресов, такая технология получила название PIC – Position Independent Code. В компиляторе gcc данная возможность включается ключом -fPIC.

Теперь компилирование наших файлов будет иметь вид:

|  |
| --- |
| nyquist:~# gcc -fPIC -c f1.c  nyquist:~# gcc -fPIC -c f2.c |

Динамическая библиотека это уже не архивный файл, а настоящая загружаемая программа, поэтому созданием динамических библиотек занимается сам компилятор gcc. Для того, чтобы создать динамическую библиотеку надо использовать ключ **-**shared:

|  |
| --- |
| nyquist:~# gcc -shared -o libfsdyn.so f1.o f2.o |

В результате получим динамическую библиотеку **libfsdyn.so**, которая будет динамической версией библиотеки **libfs.a** (что видно из названия) Теперь, чтобы компилировать результирующий файл с использованием динамической библиотеки, нам надо собрать файл командой:

|  |
| --- |
| nyquist:~# gcc -с main.с  nyquist:~# gcc main.o -L. -lfsdyn -o rezultdyn |

Если теперь Вы сравните файлы, полученные при использовании статической и динамической библиотеки, то увидите, что их размеры отличаются. В данном случае файл, созданный с динамической библиотекой, занимает чуть больше места, но это лишь оттого, что программа, используемая нами, совершенно примитивная и львиную долю там занимает специальный код (crt) для использования динамических возможностей. В реальных условиях, когда используются очень большие функции размер программы с использованием динамической библиотеки значительно меньше.

Но, если же вы сейчас попробуете запустить файл rezultdyn, то получите ошибку:

|  |
| --- |
| nyquist:~# ./rezultdyn  ./rezultdyn: error in loading shared libraries: libfsdyn.so: cannot open shared object file: No such file or directory  nyquist:~# |

Это сообщение выдает загрузчик динамических библиотек (динамический линковщик – *dynamic linker*), который в нашем случае не может обнаружить библиотеку *libfsdyn.so*. Для настройки динамического линковщика существует ряд программ.

Первая программа называется ldd. Она выдает на экран список динамических библиотек, используемых в программе, и их местоположение. В качестве параметра ей сообщается название обследуемой программы. Давайте попробуем использовать ее для нашей программы *rezultdyn*:

|  |
| --- |
| nyquist:~# ldd rezultdyn  libfsdyn.so => not found  libc.so.6 => /lib/libc.so.6 (0x40016000)  /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)  nyquist:~# |

Как видите все правильно. Программа использует три библиотеки:

* **libc.so.6** - стандартную библиотеку функций языка C++;
* **ld-linux.so.2** - библиотеку динамической линковки программ **ELF** формата;
* **libfsdyn.so** - нашу динамическую библиотеку функций.

Нашу библиотеку она найти не может. Динамический линковщик ищет библиотеки только в известных ему каталогах, а каталог нашей программы ему явно неизвестен.

Для того чтобы добавить нашу директорию с библиотекой в список известных директорий, надо отредактировать файл /etc/ld.so.conf. Например, у меня этот файл состоит из таких строк:

|  |
| --- |
| nyquist:~# cat /etc/ld.so.conf  /usr/X11R6/lib  /usr/lib/qt-3.1/lib  nyquist:~# |

Во всех этих директории хранятся всеми используемые библиотеки. В этом списке нет лишь одной директории – /lib, которая сама по себе не нуждается в описании, так как она является главной. Получается, что наша библиотека станет "заметной", если поместить ее в один их этих каталогов, либо отдельно описать в отдельном каталоге. Давайте для теста опишем, добавим строку в конец файла ld.so.conf:

|  |
| --- |
| /root |

Естественно, путь зависит от того, в каком каталоге находится ваша библиотека. Теперь после этого динамический линковщик будет знать, где можно найти наш файл, но после изменения конфигурационного файла ld.so.conf необходимо, чтобы система перечитала настройки заново. Это делает программа *ldconfig*. Пробуем запустить нашу программу:

|  |
| --- |
| nyquist:~# ldconfig  nyquist:~# ./rezultdyn  f1() = 25  f2() = 10  nyquist:~# |

Если теперь Вы удалите добавленную нами строку и снова запустите ldconfig, то данные о расположении нашей библиотеки исчезнут и будет появляться та же самая ошибка.

Но описанный метод влияет на всю систему в целом и требует доступа администратора системы.

Если же вы работаете под «обычным» пользователем, то для такого случая есть другое решение. Это использование специальной переменной среды LD\_LIBRARY\_PATH, в которой перечисляются все каталоги, содержащие пользовательские динамические библиотеки. Для того чтобы установить эту переменную в командной среде bash, надо набрать всего несколько команд. Для начала посмотрим есть ли у нас такая переменная среды:

|  |
| --- |
| nyquist:~# echo $LD\_LIBRARY\_PATH |

В ответ выводится пустая строка, означающая, что такой переменной среды нет. Устанавливается она следующим образом:

|  |
| --- |
| nyquist:~# LD\_LIBRARY\_PATH=/root  nyquist:~# export LD\_LIBRARY\_PATH |

После этого программа rezultdyn будет прекрасно работать. В случае если у Вас в системе эта переменная среды уже установлена, то, чтобы не испортить ее значение, надо новый каталог прибавить к старому значению. Делается это другой командой:

|  |
| --- |
| nyquist:~# LD\_LIBRARY\_PATH=/root:${LD\_LIBRARY\_PATH}  nyquist:~# export LD\_LIBRARY\_PATH |

Если Вы обнулите эту переменную, то снова библиотека перестанет работать:

|  |
| --- |
| nyquist:~# LD\_LIBRARY\_PATH=""  nyquist:~# export LD\_LIBRARY\_PATH  nyquist:~# ./rezultdyn  ./rezultdyn: error in loading shared libraries: libfsdyn.so: cannot open shared object file: No such file or directory  nyquist:~# |

Вы также параллельно можете зайти в систему под другим пользователем или даже тем же самым, но если Вы захотите просмотреть значение LD\_LIBRARY\_PATH, то увидите ее прежнее значение. Это означает, что переменные окружения индивидуальны для каждого сеанса пользователей.

Рассмотрим еще один момент использования библиотек. Имя динамической библиотеки fsdyn выбрано специально, чтобы оно отличалось названия статической библиотеки fs. Дело в том, что если у Вас две библиотеки статическая и динамическая с одинаковыми названиями, то есть libfs.a и libfs.so, то компилятор всегда будет использовать динамическую библиотеку.

Связано это с тем, что в ключе -l задается часть имени библиотеки, а префикс lib и окончание .a или .so приставляет сам компилятор. Так вот алгоритм работы компилятора таков, что, если есть динамическая библиотека, то она используется по умолчанию. Статическая же библиотека используется, когда компилятор не может обнаружить файл .so этой библиотеки. Вообще-то, если использовать ключ компилятора -static, то можно насильно заставить компилятор использовать статическую библиотеку:

|  |
| --- |
| nyquist:~# gcc -staticmain.o -L. -lfs -o rez1 |

Но тут возникает один нюанс – результирующий файл rez1 получается размером в 900 Кб. После применения программы strip размер ее уменьшается до 200 Кб, но это же не сравнить с тем, что наша первая статическая компиляция давала программу размером 10 Кб. А связано это с тем, что любая программа, написанная на C/C++, в использует стандартную библиотеку «C» library, которая содержит в себе определения таких функций, как printf(), write() и всех остальных. Эта библиотека линкуется к файлу как динамическая, чтобы все программы, написанные на C++, могли использовать единожды загруженные функции. Ну, а при указании ключа -static компилятор делает линковку libc статической, поэтому размер кода увеличивается на все 200 Кб.

В общем случае нюансов с библиотеками достаточно много. Поэтому для упрощения работы имена динамическим и статическим библиотеками лучше давать разные.

## Позднее связывание

Оказывается, что использовать динамические библиотеки можно не только в начале загрузки, но и в процессе самой работы программы. Программа сама может вызывать любые функции из библиотеки, когда ей захочется. Для этого всего лишь надо использовать библиотеку dl, которая позволяет линковать библиотеки «на лету». Она управляет загрузкой динамических библиотек, вызовом функций из них и выгрузкой после конца работы.

Для использования функций программной работы с динамическими библиотеками необходимо подключить заголовочный файл:

|  |
| --- |
| #include <dlfcn.h> |

Чтобы вызывать какие-то функции из динамической библиотеки сначала надо открыть эту библиотеку (можно сказать "загрузить"). Открывается она функцией:

|  |
| --- |
| void \*dlopen (const char \*filename, int flag); |

Параметр filename содержит путь до требуемой библиотеки, а параметр flag задает некоторые специфические флаги для работы. Функция возвращает указатель на загруженную библиотеку. В случае любой ошибки возвращается указатель NULL. В таком случае тест ошибки понятный человеку можно получить с помощью функции dlerror(). Ну пока мы не будем на этом акцентировать внимания. Ниже приведен «стандартный» код для открытия библиотеки:

|  |
| --- |
| void \*library\_handler;  //......  //загрузка библиотеки  library\_handler = dlopen("/path/to/the/library.so", RTLD\_LAZY);  if (!library\_handler){  //если ошибка, то вывести ее на экран  fprintf(stderr,"dlopen() error: %s\n", dlerror());  exit(1); // в случае ошибки можно, например, закончить работу программы  }; |

После этого можно работать с библиотекой. А работа эта заключается в получении адреса требуемой функции из библиотеки. Получить адрес функции или переменной можно по ее имени с помощью функции:

|  |
| --- |
| void \*dlsym(void \*handle, char \*symbol); |

Для этой функции требуется адрес загруженной библиотеки handle, полученный при открытии функцией dlopen(). Требуемая функция или переменная задается своим именем в переменной symbol.

Закрывается библиотека функцией:

|  |
| --- |
| dlclose(void \*handle); |

При закрытии библиотеки динамический линковщик проверяет счетчик количества открытий библиотеки, и если она была открыта несколькими программами одновременно, то она не выгружается до тех пор, пока все программы не закроют эту библиотеку.

Для примера создадим программу, которая в качестве параметра получает название функции, которую она будет использовать в работе. Например, это будут математические функции возведения в степень. Создадим сначала динамическую библиотеку.

|  |
| --- |
| double power2(double x)  {  return x\*x;  };  double power3(double x)  {  return x\*x\*x;  };  double power4(double x)  {  return power2(x)\*power2(x);  };  //...... |

Сохраняем его в файл lib.c и создаем динамическую библиотеку libpowers.so следующими командами:

|  |
| --- |
| nyquist:~# gcc -fPIC -c lib.c  nyquist:~# gcc -shared lib.o -o libpowers.so |

Теперь создаем основную программу в файле main.c.

|  |
| --- |
| #include <stdio.h>  /\* заголовочный файл для работы с динамическими библиотеками \*/  #include <dlfcn.h>  int main(int argc, char\* argv[])  {  void \*ext\_library; // хандлер внешней библиотеки  double value=0; // значение для теста  double (\*powerfunc)(double x);// переменная для хранения адреса функции  //загрузка библиотеки  ext\_library = dlopen("/root/libpowers.so",RTLD\_LAZY);  if (!ext\_library)  {  //если ошибка, то вывести ее на экран  fprintf(stderr,"dlopen() error: %s\n", dlerror());  return 1;  };  //загружаем из библиотеки требуемую процедуру  powerfunc = dlsym(ext\_library, argv[1]);  value=3.0;  //выводим результат работы процедуры  printf("%s(%f) = %f\n",argv[1],value,(\*powerfunc) (value));  //закрываем библиотеку  dlclose(ext\_library);  }; |

Код главной программы готов. Требуется его откомпилировать с использованием библиотеки dl:

|  |
| --- |
| nyquist:~# gcc main.c -o main -ldl |

Получим программный файл main, который можно тестировать. Наша программа должна возводить значение 3.0 в требуемую нами степень, которая задается названием функции. Пример:

|  |
| --- |
| nyquist:~# ./main  power2 power2(3.000000) = 9.000000  nyquist:~# ./main power3  power3(3.000000) = 27.000000  nyquist:~# ./main power4  power4(3.000000) = 81.000000  nyquist:~# |

# ИНДИВИДУАЛЬНЫЕ ЗАДАНИЯ

В этом разделе приведены задания, в каждом из которых требуется:

* написать программу для Unix на языке C, состоящую минимум из 2-х модулей с соответствующими заголовочными файлами;
* создать Makefile для компиляции, сборки, очистки (clean) и запуска (run) готовой программы.

Задания сопровождаются примерами использования готовых программ в командной оболочке. Студенту предлагается выполнить одно из заданий и продемонстрировать разработанную программу преподавателю.

1. Вывод информации о характеристиках центрального процессора (например, из файла /proc/cpuinfo).

|  |
| --- |
| $ ./cpuinfo –model  Intel(R) Core(TM) i5 CPU M 430 @ 2.27GHz  $ ./cpuinfo --count |

1. Загрузка файла с указанного HTTP-адреса с опцией поддержки прокси-сервера.

|  |
| --- |
| $ ./httpget -p myproxy.local:8080 http://iipo.tu-bryansk.ru/  <html>  ...  </html>  $ ./httpget http://iipo.tu-bryansk.ru/ > file.html |

1. Загрузка файла с указанного FTP-адреса с опцией поддержки прокси-сервера.

|  |
| --- |
| $ ./ftpget -p proxy:21 ftp://iipo.tu-bryansk.ru/pub/readme.txt  $ cat readme.txt  ... |

1. Поиск в указанной директории файлов, содержащих указанную подстроку, с опцией реверсивного поиска (вывод имен файлов, не содержащих заданную подстроку).

|  |
| --- |
| $ ./search 'root' /etc  passwd  services  $ ./search -n 'root'  /etc  fstab  ... |

1. Поиск заданной подстроки в оперативной памяти с выводом смещения и контекста.

|  |
| --- |
| $ ./memsearch 'root'  348: /root/.cshrc  10834: chroot  ... |

1. Генерирование случайного пароля с заданными в аргументах параметрами (длина, диапазон символов).

|  |
| --- |
| 1. $ genpass -l 8 -a "abcdef1234567890" 2. 9b36ce14 3. $ genpass 4. lkj73jhd1d |

1. Подсчет контрольной суммы указанных файлов (достаточно простого 32-битного суммирования).

|  |
| --- |
| $ ctlsum file.txt  64758463  $ ctlsum /etc/\*  /etc/fstab: 4327676  /etc/fstab.bak: 4327676  /etc/passwd: 93234428 |

1. Вывод информации о текущем состоянии центрального процессора (например, из файла /proc/stat).

|  |
| --- |
| $ ./cpustat --all  User: 7.3%  System: 2.0%  Idle: 90.7% |

1. Вывод списка процессов на экран (упрощенный аналог утилиты ps) с опцией расширенного (подробного) вывода.

|  |
| --- |
| $ ./ps  bash  ps  $ ./ps -l  9950 pts/1 00:00:02 bash  10004 pts/1 00:00:00 ps |

1. Отправка электронного сообщения по протоколу SMTP (адрес получателя, тема сообщения задаются аргументами, а текст подается на стандартный ввод).

|  |
| --- |
| $ sendmail -s "First mail" test@email.com  Текст сообщения  ^D  $ echo 'Hello from "`whoami`" | sendmail -s "Hi" test@email.com |

1. Вывод содержимого указанных файлов (аналог утилиты cat).

|  |
| --- |
| $ ./cat file.txt  Текст  $ ./cat \*.txt  Текст  ... |

1. Вывод содержимого указанной директории (упрощенный вариант утилиты ls) с опцией расширенного (подробного) вывода.

|  |
| --- |
| $ ./ls  file1.txt  file2.txt  $ ./ls -l  -rw-r--r-- 1 user group 90 апр. 21 16:25 file1.txt  -rw-r--r-- 1 user group 102 апр. 21 16:27 file2.txt |

1. Удаление файлов с поддержкой рекурсивного режима (упрощенный аналог утилиты rm).

|  |
| --- |
| $ mkdir /tmp/test /tmp/test/t1 /tmp/test/t2  $ ./rm /tmp/test  Невозможно удалить каталог «/tmp/test»  $ ./rm -R /tmp/test  $ ls /tmp/test  Нет такого файла или каталога |

1. Выделение колонок в тексте (аналог утилиты cut) с возможностью задания разделителя и указания номера колонки.

|  |
| --- |
| $ ./cut -d ':' -f 1 /etc/passwd  root  daemon  ... |

1. Подсчет количества символов, слов или строк в указанных файлах (аналог утилиты wc).

|  |
| --- |
| $ ls | ./wc -l  2  $ ./wc -l \*.txt  2 3 15 file1.txt  1 1 5 file2.txt |

1. Вывод аргументов команды на экран с поддержкой отмены символа перевода строки (аналог утилиты echo).

|  |
| --- |
| $ ./echo hello  hello  $ ./echo "Hello, $USER"  Hello, guest |

1. Построчное объединение содержимого файлов (упрощенный аналог утилиты paste).

|  |
| --- |
| $ ls > file1.txt  $ ls > file2.txt  $ ./paste file?.txt  file1 txt file1.txt  file2.txt |

1. Вывод текущей даты с возможность задания формата (упрощенный аналог утилиты date).

|  |
| --- |
| $ ./date  Чт. апр. 3 17:41:38 MSK 2014  $ ./date "+%Y-%m-%d %H:%M:%S"  2014-04-03 17:43:20 |

1. Копирование файлов (упрощенный аналог утилиты cp).

|  |
| --- |
| $ ./cp file1.txt file2.txt  $ ./cp file?.txt /tmp/ |

1. Вывод первых строк файла с возможностью задания количества строк (упрощенный аналог утилиты head).

|  |
| --- |
| $ ./head -n 1 /etc/passwd  root:x:0:0:root:/root:/bin/bash |

1. Вывод последних строк файла с возможностью задания количества строк (упрощенный аналог утилиты tail).

|  |
| --- |
| $ ./tail -n 1 /etc/passwd  user:x:1000:1000:user:/home/user:/bin/bash |

1. Замена или удаление символов в указанных файлах или стандартном вводе (упрощенный аналог утилиты tr).

|  |
| --- |
| $ pwd | tr / ' '  home user  $ pwd | tr -d /  homeuser |

1. Выполнение арифметических операций +, -, /, \* над аргументами (упрощенный аналог утилиты expr).

|  |
| --- |
| $ ./expr 2 + 2 / 2  3 |

1. Проверка существования файлов и каталогов (упрощенный аналог утилиты test).

|  |
| --- |
| $ ./test -f /etc/passwd && echo "файл существует"  файл существует  $ ./test -d /etc && echo "каталог существует"  каталог существует |

1. Удаление повторяющихся строк в указанных файлах или стандартном вводе (упрощенный аналог утилиты uniq).

|  |
| --- |
| $ ls \*.\* | cut -f 2 -d . | ./uniq  txt |

1. Копирование файлов (упрощенный аналог утилиты mv).

|  |
| --- |
| $ ./mv file2.txt file3.txt  $ ./mv file?.txt /tmp/ |

1. Изменение времени доступа и модификации файла (упрощенный аналог утилиты touch).

|  |
| --- |
| $ ./touch file1.txt file2. txt  $ ./touch file\*.txt |

# Контрольные вопросы

1. Как в программе осуществляется работа со стандартными вводом и выводом?
2. Каким образом аргументы командной строки интерпретируются в программе?
3. В чем назначение утилиты make?
4. Что такое Makefile и какова его структура?
5. Какова последовательность компилирования и сборки программы с использованием утилиты make?

# Список рекомендуемой литературы

## Основная литература

1. Курячий Г.В. Операционная система UNIX: учебное пособие [Электронный ресурс] / Курячий Г.В.— М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. – 258c. – Режим доступа: http://www.iprbookshop.ru/52199.
2. Администрирование ОС Unix [Электронный ресурс] / – 2-е изд. –М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. – 303 c. – Режим доступа: http://www.iprbookshop.ru/73659.html.

## Дополнительная литература

1. Командная строка UNIX: практикум [Электронный ресурс] — М.: Московский государственный строительный университет, ЭБС АСВ, 2013. – 44c. – Режим доступа: http://www.iprbookshop.ru/23729.
2. Мошков, М.Е. Введение в системное администрирование Unix [Электронный ресурс] / М.Е. Мошков. – М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. – 208 c. – Режим доступа: http://www.iprbookshop.ru/73672.html.

UNIX-системы. Программирование под UNIX на языке C: методические указания к выполнению лабораторной работы № 5 для студентов очной формы обучения по по направлениям подготовки 09.03.01 «Информатика и вычислительная техника», 09.03.04 «Программная инженерия», 02.03.03 «Математическое обеспечение и администрирование информационных систем»

ТРУБАКОВ ЕВГЕНИЙ ОЛЕГОВИЧ

Научный редактор Д. А. Коростелев

Компьютерный набор Е.О. Трубаков

Иллюстрации Е.О. Трубаков

Подписано в печать \_\_.\_\_.\_\_. Усл.печ.л. 1,8 Уч.-изд.л. 1,8

Брянский государственный технический университет

241035, Брянск, бульвар 50 лет Октября, 7 БГТУ

Кафедра «Информатика и программное обеспечение», тел. 56-09-84