2022

Лабораторная работа #4

Создание и использование библиотек в Linux

(Краткая теория, задания)

РАЗРАБОТАЛ

СЕРГЕЙ СТАНКЕВИЧ (SERHEY STANKEWICH, MINSK, BELARUS)

Operating Systems and System Programming

ЛАБОРАТОРНАЯ РАБОТА #4

Создание и использование библиотек в Linux.

Цель работы

Изучить и закрепить на практике создание и использование статически и динамически подключаемых библиотек в операционных системах семейства Linux.

Норма времени выполнения: 6 академически часов.
Оценка работы: 9 зачетных единиц.

Краткие теоретические сведения.

Библиотеки

Программы редко действуют в одиночку; чаще они полагаются на наличие других программных компонентов. Это избавляет программиста от необходимости создавать то, что уже создано. Такими компонентами могут быть *программы-библиотеки*. Библиотеки позволяют разным программам использовать один и тот же объектный код.

Библиотека (*library*) – это набор соединенных определенным образом объектных файлов.

Библиотеки подразделяются на две категории:

- статические (архивы);
- динамические (совместно используемые).

Статические библиотеки (*static libraries*) создаются программой **ar**. Файлы статических библиотек имеют расширение **.a**. Если, например, статическая библиотека foo (представленная файлом libfoo.a) создана из двух объектных файлов foo1.o и foo2.o, то следующие две команды будут эквивалентны:

```
$ gcc -o myprogram myprogram.o foo1.o foo2.o
$ gcc -o myprogram myprogram.o -lfoo
```

Совместно используемые библиотеки (*shared libraries*). Программы редко действуют в одиночку; чаще они полагаются на наличие других программных

компонентов. Например, стандартные операции, такие как ввод/вывод, выполняются процедурами, которые совместно используются многими программами. Эти процедуры хранятся в так называемых *разделяемых библиотеках* (*shared libraries*), предоставляющих важные услуги нескольким программам.

Если программе требуется некий общий ресурс, такой как разделяемая библиотека, про него говорят, что он имеет ***зависимость***.

Совместно используемые библиотеки, их также можно называть *динамическими*, не помещают свой код непосредственно в программу, а лишь создают *специальные ссылки*. Поэтому любая программа, скомпонованная с *динамической библиотекой*, при запуске требует наличия данной библиотеки в системе.

Совместно используемые библиотеки создаются компоновщиком при вызове gcc с опцией -shared и имеют расширение .so.

Заголовочные файлы как интерфейсы библиотек. Иногда *заголовочные файлы* тоже называют библиотеками. Это не так! Библиотеки – это объектный код, сгруппированный таким образом, чтобы им могли пользоваться разные программы. Заголовочные файлы – это часть исходного кода программы, в которых, как правило, определяются соглашения по использованию общих идентификаторов (имен). Когда в заголовочном файле определены механизмы, реализованные в библиотеке, то правильно будет называть такой файл (или группу файлов) *интерфейсом библиотеки*.

Создание библиотек

Создание статических библиотек. Статическая библиотека – это архив, создаваемый специальным архиватором **ar** из пакета GNU binutils. Утилита **ar** создает архив, который может подключаться к программе во время компоновки на правах библиотеки.

Рассмотрим некоторые опции архиватора. Опция **r** создает архив, а также добавляет или заменяет файлы в существующем архиве. Например, если нужно создать статическую библиотеку libfoo.a из файлов foo1.o и foo2.o, то для этого достаточно ввести следующую команду:

```
$ ar r libfoo.a foo1.o foo2.o
```

Опция **x** извлекает файлы из архива:

```
$ ar x libfoo.a
```

Опция **c** приостанавливает вывод сообщений о том, что создается библиотека:

```
$ ar cr libfoo.a foo1.o foo2.o
```

Опция **v** включает режим подробных сообщений (verbose mode):

```
$ ar crv libfoo.a foo1.o foo2.o
```

Рассмотрим пример создания статической библиотеки, в которой реализуются две функции для работы с окружением. Разобьем исходный код функций на два файла: **mysetenv.c** и **myprintenv.c**.

Чтобы функции, содержащиеся в этих файлах, вызывались по правилам, нужно создать заголовочный файл **myenv.h**.

Теперь организуем автоматическую сборку библиотеки **libmyenv.a.** Для этого необходимо создать *make*-файл.

Листинг файла **Makefile**:

```
libmyenv.a: mysetenv.o myprintenv.o
	ar rv $@ $^
mysetenv.o: mysetenv.c
	gcc -c $^
myprintenv.o: myprintenv.c
	gcc -c $^
clean:
	rm -f libmyenv.a *.o
```

Затем напишем программу **envmain.c**, к которой будет подключаться полученная библиотека, то есть будут вызываться соответствующие функции пользовательских библиотек. Осталось только собрать эту программу с библиотекой **libmyenv.a**:

```
$ gcc -o myenv envmain.c -L. -lmyenv
```

Сборку программы и создание библиотеки можно объединить в один проект. Для этого создадим универсальный *make*-файл.

Листинг универсального файла **Makefile**:

```
myenv: envmain.o libmyenv.a
	gcc -o myenv envmain.o -L. -lmyenv
envmain.o: envmain.c
	gcc -c $^
libmyenv.a: mysetenv.o myprintenv.o
	ar rv $@ $^
```

```
mysetenv.o: mysetenv.c
	gcc -c $^
myprintenv.o: myprintenv.c
	gcc -c $^
clean:
	rm -f myenv libmyenv.a *.o
```

В этом варианте сборка программы **envmain** осуществляется в два этапа. В качестве главной цели теперь используется **myenv**, а цель **libmyenv.a** попала в список зависимостей.

Но это не мешает, например, собирать в рамках данного проекта только библиотеку:

```
$ make libmyenv.a

gcc -c mysetenv.c
gcc -c myprintenv.c
ar rv libmyenv.a mysetenv.o myprintenv.o
ar: creating libmyenv.a
a — mysetenv.o
a — myprintenv.o
```

Создание совместно используемых библиотек. Процесс создания и подключения совместно используемых библиотек несколько сложнее, чем статических. Динамические библиотеки создаются при помощи gcc по следующему шаблону:

```
$ gcc -shared -o LIBRARY_NAME FILE1.o FILE2.o ...
```

В действительности gcc вызывает компоновщик **ld** с опциями для создания совместно используемой библиотеки. В принципе, этот процесс внешне ничем (кроме опции -shared) не отличается от компоновки обычного исполняемого файла. Но не все так просто.

При создании динамических библиотек нужно учитывать следующее:

- в процессе компоновки совместно используемой библиотеки должны участвовать объектные файлы, содержащие ***позиционно-независимый код*** (*Position Independent Code*).;
- опция -L, указанная при компоновке, позволяет дополнить список каталогов, в которых будет выполняться поиск библиотек.

Позиционно-независимый код имеет возможность подгружаться к программе в момент ее запуска. Чтобы получить объектный файл с позиционно-независимым кодом, нужно откомпилировать исходный файл с опцией -fPIC

Как происходит поиск библиотек. По умолчанию в исполняемом файле сохраняется лишь имя библиотеки, а во время запуска программы происходит повторный поиск библиотек.

> Поэтому программист *должен учитывать месторасположение динамической библиотеки* не только на своем компьютере, *но и там, где будет впоследствии запускаться программа*.

Можно выделить четыре способа определения пути для поиска библиотек:

1. Пути определяются при конфигурировании утилиты ld во время ее компиляции.
2. Пути могут быть указаны через переменную окружения LD_LIBRARY_PATH.
3. Пути могут быть заданы в командной строке.
4. Путь можно указать и в самой программе.

Чаще всего системные библиотеки находятся в каталогах /lib и /usr/lib, поэтому два этих каталога просматриваются автоматически.

Другие каталоги задаются одной или несколькими опциями -L в командной строке. Например, по следующей команде компоновщик будет просматривать текущий каталог и каталог с именем /home/Desktop/lib для поиска любой библиотеки, не обнаруженной в пути поиска по умолчанию:

```
gcc -L. -L/home/Desktop/lib hello.o
```

В момент запуска программы для поиска библиотек просматриваются каталоги, перечисленные в файле /etc/ld.so.conf и в переменной окружения LD_LIBRARY_PATH. Синтаксис файла /etc/ld.so.conf позволяет при помощи инструкции include подгружать содержимое других файлов, содержащих информацию о расположении библиотек.

Переменная окружения LD_LIBRARY_PATH имеет тот же формат, что и переменная PATH, т. е. содержит список каталогов, разделенных двоеточиями. Известно, что окружение нестабильно и может изменяться в ходе наследования от процесса к процессу. Поэтому использование LD_LIBRARY_PATH — не самый разумный ход.

В процессе компоновки программы можно отдельно указать каталог, где будет размещаться библиотека. Для этого линковщику ld необходимо передать опцию -rpath при помощи опции -Wl компилятора gcc. Например, чтобы занести в исполняемый файл prog месторасположение библиотеки **libfoo.so**, нужно сделать следующее:

```
$ gcc -o prog prog.o -L./lib/foo -lfoo -Wl,-rpath,/lib/foo
```

Итак, опция -Wl сообщает gcc о необходимости передать линковщику определенную опцию. Далее, после запятой, следует сама опция и ее аргументы, также разделенные запятыми. Такой подход выглядит лучше, чем применение LD_LIBRARY_PATH, однако и здесь есть существенный недостаток.

> Нет никаких гарантий, что на компьютере у конечного пользователя библиотека libfoo.so будет также находиться в каталоге /lib/foo.

Есть еще один способ заставить программу искать совместно используемую библиотеку в нужном месте. Во время инсталляции программы можно добавить запись с каталогом месторасположения библиотеки в файл /etc/ld.so.conf либо в один из файлов, добавленных в ld.so.conf инструкцией include. Но это делают крайне редко, поскольку слишком длинный список каталогов в этом файле может отразиться на скорости загрузки системы. Обычно к такому подходу прибегают только такие "именитые" проекты, как Qt или X11. Многие Linux-системы при загрузке читают файл /etc/ld.so.conf и создают кэш динамических библиотек.

Наилучший выход из сложившегося положения – размещать библиотеки в специально предназначенных для этого каталогах (/usr/lib или /usr/local/lib). Естественно, программист в ходе работы над проектом может для удобства пользоваться переменной LD_LIBRARY_PATH или опциями -Wl и -rpath, но в конечной программе лучше избегать этих приемов и просто располагать библиотеки в обозначенных ранее каталогах.

Как это делается на практике. За основу возьмем пример из предыдущего проекта.

Рассмотрим сначала концепцию использования *переменной окружения* LD_LIBRARY_PATH. Чтобы переделать предыдущий пример для работы с динамической библиотекой, требуется лишь изменить ранее созданный make-файл.

Листинг модифицированного файла **Makefile**:

```
myenv: envmain.o libmyenv.so
	gcc -o myenv envmain.o -L. -lmyenv
envmain.o: envmain.c
	gcc -c $^
libmyenv.so: mysetenv.o myprintenv.o
	gcc -shared -o libmyenv.so $^
mysetenv.o: mysetenv.c
	gcc -fPIC -c $^
myprintenv.o: myprintenv.c
	gcc -fPIC -c $^
clean:
	rm -f myenv libmyenv.so *.o
```

Обратите внимание, что файлы mysetenv.o и myprintenv.o, участвующие в создании библиотеки, компилируются с опцией -fPIC для генерирования позиционно-независимого кода. Файл envmain.o не добавляется в библиотеку, поэтому он компилируется без опции -fPIC.

Если теперь попытаться запустить исполняемый файл myenv, то будет выдано сообщение об ошибке:

```
$ ./myenv MYVAR Hello
./myenv: error while loading shared libraries: libmyenv.so:
cannot open shared object file: No such file or directory
```

Проблема в том, что программа не нашла библиотеку в стандартном списке каталогов. После установки переменной LD_LIBRARY_PATH проблема исчезнет:

```
$ export LD_LIBRARY_PATH=.
$ ./myenv MYVAR Hello
Setting variable MYVAR
MYVAR=Hello
```

Если подняться в родительский каталог и попытаться оттуда запустить программу **myenv**, то опять будет обнаружена ошибка:

```
$ cd ..
$ myenv/myenv MYVAR Hello
myenv/myenv: error while loading shared libraries: libmyenv.so:
cannot open shared object file: No such file or directory
```

Очевидно, что ошибка произошла из-за того, что в текущем каталоге не нашлась требуемая библиотека. Этот поучительный пример говорит о том, что в LD_LIBRARY_PATH лучше заносить абсолютные имена каталогов, а не относительные.

Попробуем еще раз:

```
$ cd myenv
$ export LD_LIBRARY_PATH=$PWD
$ cd ..
$ myenv/myenv MYVAR Hello
Setting variable MYVAR
MYVAR=Hello
```

Переменная окружения PWD содержит абсолютный путь к текущему каталогу, а запись $PWD подставляет это значение в команду.

Попробуем теперь указать линковщику опцию -rpath. Для этого изменим в make-файле первую целевую связку:

```
myenv: envmain.o libmyenv.so
	gcc -o myenv envmain.o -L. -lmyenv -Wl,-rpath,.
```

Помимо этого, для чистоты эксперимента удалим из окружения переменную LD_LIBRARY_PATH:

```
$ unset LD_LIBRARY_PATH
```

Теперь программа запускается из любого каталога без манипуляций с окружением.

Подключение библиотек

Библиотеки подключаются к программе на стадии компоновки. Например, функция printf() реализована в стандартной библиотеке языка C, которая автоматически подключается к программе, когда компоновка осуществляется посредством gcc.

Чтобы подключить библиотеку к программе, нужно передать компоновщику опцию -l, указав после нее (можно без разделяющего пробела) имя библиотеки. Если, например, к программе необходимо подключить библиотеку **mylibrary**, то для осуществления компоновки следует задать такую команду:

```
$ gcc -o myprogram myprogram.o -lmylibrary
```

Чаще всего файлы библиотек располагаются в специальных каталогах (/lib, /usr/lib, /usr/local/lib и т.п.). Если же требуется подключить библиотеку из другого места, то при компоновке следует указать опцию -L, после которой (можно без разделяющего пробела) указывается нужный каталог. Таким образом, при необходимости подключения библиотеки mylibrary, находящейся в каталоге /usr/lib/particular, для выполнения компоновки указывают такую команду:

```
$ gcc -o myprogram myprogram.o -L/usr/lib/particular -lmylibrary
```

Следует отметить, что имена файлов библиотек обычно начинаются с префикса lib.

> Имя библиотеки получается из имени файла отбрасыванием префикса lib и расширения. Если, например, файл библиотеки имеет имя libmylibrary.so, то сама библиотека будет называться mylibrary.

Рассмотрим в качестве примера программу для возведения числа в степень.

Листинг программы **power.c**:

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int main (int argc, char ** argv)
{
        if (argc < 3) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }
        printf ("%f\n", pow (atof (argv[1]), atof (argv[2])));
        return 0;
}
```

Функции **printf()** и **atof()** реализованы в стандартной библиотеке языка C, которая автоматически подключается к проекту во время компоновки. Функция **pow()** принадлежит библиотеке математических функций, которую нужно подключать отдельно. Математическая библиотека, представленная файлами **libm.so** (*динамический* вариант) и **libm.a** (*статический* вариант), имеется

практически в любой Linux-системе. Следовательно, для сборки приведенной программы необходимо указать следующую команду:

```
$ gcc -o power1 power.c -lm
```

Возникает вопрос: какая именно библиотека была подключена? Оба варианта математической библиотеки (статическая и динамическая) подходят под шаблон -lm. В таких ситуациях предпочтение отдается динамическим библиотекам. Но если при компоновке указать опцию -static, то приоритет изменится в сторону статической библиотеки:

```
$ gcc -static -o power2 power.c -lm
```

Чтобы использовать статический вариант математической библиотеки, в вашей Linux-системе должен быть установлен пакет **glibc-static-devel**.

Теперь сравните размеры исполняемых файлов. Бинарник, полученный в результате линковки с опцией -static, значительно больше:

```
-rwxr-xr-x 1 nnivanov nnivanov 5.9K 2011-04-29 07:14 power1*
-rwxr-xr-x 1 nnivanov nnivanov 615K 2011-04-29 07:20 power2*
```

Это обусловлено тем, что статическая библиотека полностью внедряется в исполняемый файл, а совместно используемая библиотека лишь оставляет информацию о себе.

Вызов библиотеки

Динамический библиотеки могут вызываться (подгружаться) двумя способами: *статическим* и *динамическим*. Статический способ подгрузки библиотеки мы рассмотрели выше. Здесь мы рассмотрим динамический вызов динамической библиотеки. Этот способ вызова в некотором роде проще чем статический вызов.

Суть *динамической подгрузки* состоит в том, что запущенная программа может по собственному усмотрению подключить к себе какую-либо (динамическую) библиотеку. Благодаря этой возможности создаются программы с подключаемыми плагинами, такие как XMMS. Для этого используется функция **dlopen()**.

Вызов dlopen(), загружает в память динамическую (разделяемую) библиотеку (если она еще не загружена) и возвращает идентификатор, используемый для адресации к ее функциям.

Вызов dlsym() возвращает адреса функций, которые потом могут вызываться, как будто бы они находятся в главной программе main.

Функция dlclose() отсоединяет (detache) главную (текущую) программу от загруженной разделяемой библиотеки (что ОЧЕНЬ удобно, если необходимо минимизировать объем используемой оперативной памяти). Отметим, что если к динамической библиотеке не присоединено больше ни одной программы, то она выгружается из памяти.

Функция dlerror() возвращает строку описания ошибки, произошедшей при последнем вызове одной из функций dlopen(), dlclose(), dlsym(). При отсутствии ошибок dlerror() возвращает значение NULL.

Второй аргумент вызова функции dlopen() — флаг способа динамической загрузки библиотеки. Он может иметь следующие значения. При значении RTLD_NOW все функции библиотеки сразу загружаются в память и после этого становятся доступными для вызова. При значении флага RTLD_LAZY загрузка каждой функции задерживается до тех пор, пока ее имя не будет передано функции dlsym(). Каждое из двух этих значений флага может быть соединено с помощью ключевого слова OR со значением RTLD_GLOBAL. При этом все внешние вызовы загружаемой динамической библиотеки разрешаются вызовом функций из других динамических библиотек. Последние при этом также загружаются в (оперативную) память компьютера.

Практический пример создания и подключения динамической библиотеки динамическим способом представлен в *упражнении №2*.

Взаимодействие библиотек

Бывают случаи, когда динамическая библиотека сама компонуется с другой библиотекой. В этом нет ничего необычного. Такие зависимости между библиотеками можно увидеть при помощи программы **ldd**. Даже библиотека libmyenv.so из предыдущего раздела автоматически скомпонована со стандартной библиотекой языка C:

```
$ ldd libmyenv.so
linux-gate.so.1 =>  (0xffffe000)
libc.so.6 => /lib/i686/libc.so.6 (0xb76c0000)
/lib/ld-linux.so.2 (0xb7833000)
```

Статические библиотеки не могут иметь таких зависимостей, но это не мешает им участвовать в создании динамических библиотек.

Практический пример создания динамической библиотеки с использованием других библиотек представлен в *упражнении №3*.

Вывод списка зависимостей, связанных с динамическими библиотеками при помощи утилиты ldd

Для статической библиотеки

Предположим, мы имеем исполняемый файл main. Запустим утилиту **ldd**.

```
$ ldd main
```

Примерно вот что получится:

```
linux-vdso.so.1 =>  (0x00007ffc9bb33000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f5f14039000)
/lib64/ld-linux-x86-64.so.2 (0x00007f5f14403000)
```

Как можно заметить здесь есть лишь зависимости, связанные с системными библиотеками Linux. Самой же статической библиотеки, естественно, нет, так как она вошла в состав исполняемого файла. В этом можно косвенно убедиться, сравнив объемы файлов. Посмотрите, сколько байтов составляет сумма объема объектные файлы проекта, и сравните ее с объемом исполняемого файла main.

Вы можете заметить, что объем исполняемого файла немного больше. Скорее всего, это за счет наличия разного рода управляющей информации, определения адресов расположения присоединенных к нему функций и т.д.

Для динамической библиотеки, подключенной *статическим* образом

```
$ ldd main1
```

Получим:

```
linux-vdso.so.1 =>  (0x00007ffce59fb000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fadc5e65000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fadc5a9b000)
/lib64/ld-linux-x86-64.so.2 (0x00007fadc6069000)
```

Как видим, во второй строчке присутствует пользовательская библиотека libmain1.so, и указан ее начальный виртуальный адрес.

Для динамической библиотеки, подключенной *динамическим* образом

```
$ ldd main2
```

Получим:

```
linux-vdso.so.1 =>  (0x00007ffce59fb000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fadc5e65000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fadc5a9b000)
/lib64/ld-linux-x86-64.so.2 (0x00007fadc6069000)
```

Любопытно, что при динамической подгрузке в перечне зависимостей нет нашей пользовательской библиотеки libFunc1_Func2.so, упомянутой в исходном коде **main2.c**. А почему? Потому, что эту библиотеку мы не собирали вместе главной функцией, а лишь сделали ее динамическую подгрузку. Поэтому компилятор и не включил эту библиотеку в перечень зависимостей. Вместо нее фигурирует системная библиотека libdl.so.2. Для подключения этой библиотеки и используется опция -ldl.

Best of LUCK with it, and remember to HAVE FUN while you're learning :)
Sergey Stankewich

УПРАЖНЕНИЯ

Упражнение 1

Изучите и выполните требования, представленные в лабораторной работе №5 «Создание и использование библиотек в Linux».

Файлы с содержанием лабораторной работы предоставлены.

Упражнение 2

Давайте создадим пару библиотек, которые будем потом подключать разными способами:

```
1.  /* function1.c */
2.  #include <stdio.h>
3.  void function1() {
```

```
4.  printf("\tLOADED Function 1 worked...\n");
5.  }
```

```
1.  /* function2.c */
2.  #include <stdio.h>
3.  void function2(char *string) {
4.  printf("%s\n", string);
5.  }
```

Т.е. это – обычные функции типа «Hello World».

Заголовочный файл main.h может иметь следующий вид:

```
1.  /* main.h */
2.  void function1 (void);
3.  void function2 (void);
```

Далее рассмотрим главную функцию (файл main2.c), которая, собственно, и будет подгружать эту динамическую библиотеку и вызывать имеющиеся в ней функции function1() и function2():

```
1.  /* main2.c */
2.  // gcc main2.c -ldl -o main2 -Wl,-rpath,.
3.  #include <dlfcn.h>
4.  #include <stdio.h>
5.  #include <stdlib.h>

6.  int main(int argc,char *argv[]) {
7.  void *LIB_so;
8.  char *error;
9.  void (*funct1)(void);
10. void (*funct2)(char *);
11. LIB_so = dlopen ("libFunc1_Func2.so",
RTLD_LAZY);
12. //  LIB_so  =  dlopen  ("./libFunc1_Func2.so",
RTLD_LAZY);
13. // Если не указывать опцию -Wl,-rpath,. и если
libFunc1_Func2.so расположена в текущем каталоге
14. if (error = dlerror()) {
15. printf("dlopen: %s\n",error);
16. exit(1);
17. }

18. funct1 = dlsym(LIB_so, "function1");
19. if (error = dlerror()) {
20. printf("dlsym1: %s\n",error);
```

```
21.  exit(1);
22.  }

23.  funct2 = dlsym(LIB_so, "function2");
24.  if (error = dlerror()) {
25.  printf("dlsym2: %s\n", error);
26.  exit(1);
27.  }

27.  printf("Testing Function1: \n");
28.  funct1();
29.  printf("Testing Function2: \n");
30.  funct2("\tLOADED Function 2 worked.");
31.  dlclose(LIB_so);
32.  }
```

Как видим, при той же самой функциональности ее исходный код несколько усложнился – добавились строчки, обрабатывающие динамическую подгрузку библиотеки.

Компилировать следует так:

```
gcc main2.c -ldl -o main2 -Wl,-rpath,.
```

Получим состав файлов проекта:

- function1.c
- function2.c
- libFunc1_Func2.so
- main2.c
- main2

> Внимание! Возможны некоторые проблемы с кодом! Лицензия *GPL* не дает гарантии.
>
> Попробуйте поработать с указателями и синтаксическим определением функций. Изучите и используйте утилиту **nm**.

Когда проблемы с кодом будут устранены, готовую программу запустим – как обычно:

```
./main2
```

Таким образом, данный способ (динамической подгрузки) дает очень даже удобные и легкие возможности по использованию библиотек в собственных программах: достаточно лишь указать их имена и затем подгрузить при помощи функции dlopen(). После чего останется лишь вызывать содержащиеся в библиотеках функции.

Обратите внимание: в отличие от первого варианта, здесь, при динамической подгрузке, НЕТ НЕОБХОДИМОСТИ собирать динамическую библиотеку вместе с объектным файлом программы main2.o – в отличие от первого варианта («обычной» компиляции динамической библиотеки).

Вот видите, как все просто и удобно.

Упражнение 3

Рассмотрим практический пример создания динамической библиотеки с использованием других библиотек. Для этого нужно создать четыре исходника (по одному на каждую библиотеку и еще один – для исполняемой программы).

Исходный файл **first.c**:

```
#include <stdio.h>
#include "common.h"

void do_first (void) { printf ("First library\n"); }
```

Исходный файл **second.c**:

```
#include <stdio.h>
#include "common.h"

void do_second (void) { printf ("Second library\n"); }
```

Исходный файл **third.c**:

```
#include "common.h"

void do_third (void)
{
        do_first ();
        do_second ();
}
```

Общий для всех файл common.h содержит объявления библиотечных функций:

```
#ifndef COMMON_H
#define COMMON_H

void do_first (void);
void do_second (void);
void do_third (void);

#endif
```

Программа **program.c**:

```
#include "common.h"

int main (void)
{
        do_third ();
        return 0;
}
```

Теперь создадим make-файл, который откомпилирует все исходные файлы и создаст три библиотеки, одна из которых будет статической:

```
program: program.c libthird.so
        gcc -o program program.c -L. -lthird \
        -Wl,-rpath,.
libthird.so: third.o libfirst.so libsecond.a
        gcc -shared -o libthird.so third.o -L. \
        -lfirst -lsecond -Wl,-rpath,.
third.o: third.c
        gcc -c -fPIC third.c
libfirst.so: first.c
        gcc -shared -fPIC -o libfirst.so first.c
libsecond.a: second.o
        ar rv libsecond.a second.o
second.o: second.c
        gcc -c second.c
clean:
        rm -f program libfirst.so libsecond.a \
        libthird.so *.o
```

В итоге, make-файл содержит семь целевых связок.

Теперь при помощи программы **ldd** проверим зависимости, образовавшиеся между библиотеками:

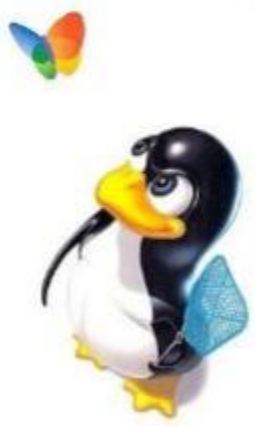
```
$ ldd libthird.so
linux-gate.so.1 =>  (0xffffe000)
libfirst.so => ./libfirst.so (0xb787c000)
libc.so.6 => /lib/i686/libc.so.6 (0xb770c000)
/lib/ld-linux.so.2 (0xb7881000)
```

Как и ожидалось, библиотека **libthird.so** зависит от **libfirst.so**. Если вызвать программу **ldd** для исполняемого файла **program**, то можно увидеть образовавшуюся цепочку зависимостей полностью:

```
$ ldd program
linux-gate.so.1 =>  (0xffffe000)
libthird.so => ./libthird.so (0xb77ef000)
libc.so.6 => /lib/i686/libc.so.6 (0xb767f000)
libfirst.so => ./libfirst.so (0xb767d000)
/lib/ld-linux.so.2 (0xb77f2000)
```

Очевидно, что архив **libsecond.a** никак не фигурирует в выводе команды **ldd**. Статические библиотеки никогда не образуют зависимости, поскольку для обеспечения автономной работы их код полностью включается в результирующий файл.

We hope you enjoy working with Linux!



ЗАДАНИЯ

Задание 1

Произвести рефакторинг проекта предыдущей лабораторной работы по автосборке проекта: вынесите код функций бизнес логики в отдельную *статическую* библиотеку. Для автосборки проекта используйте утилиту **make**, обязательно.

При помощи программы **ldd** проверим зависимости, образовавшиеся между библиотеками.

Сравните сумму размеров объектных файлов проекта и исполняемого файла.

Результаты отобразите скриншотами.

Копию исполняемого файла расположите на «Рабочем столе», запустите программу. Какой будет результат?

Задание 2

Произвести рефакторинг проекта предыдущей лабораторной работы по автосборке проекта: вынесите код функций бизнес логики в отдельную *динамическую* библиотеку со *статическим* вызовом. Для автосборки проекта используйте утилиту **make**, обязательно.

При помощи программы **ldd** проверим зависимости, образовавшиеся между библиотеками.

Сравните сумму размеров объектных файлов проекта и исполняемого файла.

Результаты отобразите скриншотами.

Копию исполняемого файла расположите на «Рабочем столе», запустите программу. Какой будет результат? Если возникла проблема, решите ее и поясните каким способом.

Сравните результаты первого и вторгог упражнения, сделайте вывод.

Задание 3

Произвести рефакторинг проекта предыдущей лабораторной работы по автосборке проекта: вынесите код функций бизнес логики в отдельную *динамическую* библиотеку с *динамическим* вызовом. Для автосборки проекта используйте утилиту **make**, обязательно.

При помощи программы **ldd** проверим зависимости, образовавшиеся между библиотеками.

Сравните сумму размеров объектных файлов проекта и исполняемого файла.

Результаты отобразите скриншотами.

Копию исполняемого файла расположите на «Рабочем столе», запустите программу. Какой будет результат? Если возникла проблема, решите ее и поясните каким способом.

Сравните результаты с предыдущими упражнения, сделайте вывод.

«Easy things should be easy and hard things should be possible»
«Простые вещи должны быть простыми, а сложные вещи должны быть возможными»

Контрольные вопросы:

1) Что такое библиотеки и для чего они используются?
2) Чем отличается заголовочные файлы (файлы с расширением *.h) от файлов библиотек?
3) Опишите существующие типы библиотек и принципы их использования.
4) Опишите преимущества и недостатки каждого из типа библиотек.
5) Как и какими способами подключить библиотеку к программе? Опишите специфику подключения.
6) На какой стадии происходит подключение библиотек (объектных файлов) к основному модулю программы?
7) Как создать статическую и динамическую библиотеки и подключить их к программе?
8) Что такое «позиционно-независимый код» (position independent code, PIC)?
9) Какие способы можно использовать для того, чтобы указать местоположение подключаемых динамических библиотек?
10) Если в целевом каталоге присутствует два типа одной и той же библиотеки, какая из низ всегда будет подключаться по умолчанию? А как явно подключить вторую?
11) Какие проблемы могут быть при обновлении совместно используемых библиотек, и что такое «Ад DLL» (DLL HELL).

Дополнительная информация

Подробоно песочница придставлена к книге: Шоттс У. «Командная строка Linux. Полное руководство.» — СПб.: Питер, 2017. — 480 с.: ил. — (Серия «Для профессионалов»). на страницах 225-226.

Робачевский А. М. Операционная система UNIX®. - СПб.: 2002. - 528 ил.

Интернет источники

Задание 6 по предмету Операционные Системы: Создание статических и динамических библиотек в Linux (4846d.ru)

Салимóненко Дмитрий Александрович.
Задание 6: Создание статических и динамических библиотек.

Справочная информация

Команда **ar** создает статическую библиотеку (архив). В данном случае два объектных файла объединяются в один файл `libmain.a`.

Опция **-I**, переданная компилятору, обрабатывается и посылается линковщику для того, чтобы тот подключил к бинарному файлу библиотеку.

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

Опция **-l** используется для реализации ссылок на библиотеки, которые существуют в каталогах, указанных в параметрах **-L**. Эти каталоги также появляются в `LD_LIBRARY_PATH`. Таким образом, получается преобразование имен библиотек:

```
1.    libmain.a -> -lmain
2.    libmain.so -> -lmain
```

Опция **-static** указывает линковщику использовать только статические версии всех необходимых приложению библиотек:

```
gcc -static -o main main.o -L. -lmain
```

Опция `-shared` вызывает линковщик, только не для сборки исполняемого файла, а для создания динамической библиотеки.

Опция `-fPIC` генерирует позиционно-независимый код для динамических библиотек (если используется опция -shared).

Опция `-fpic` создает более эффективный код (если поддерживается платформой компилятора).

Опция `-Wl,` используется для передачи линковщику (сборщику) каких-либо своих (дополнительных) опций. Например, так можно передать путь к библиотеке.

Опция `-rpath,.` необходима для указания линковщику пути к библиотеке (в данном случае – текущий каталог). Передается при помощи опции `-Wl,`

Опция `-fPIC` (`-fpic`) при компиляции `function1.c` и `function2.c` сообщает компилятору, что объектные файлы, полученные в результате компиляции должны содержать позиционно-независимый код (PIC - Position Independent Code), который используется в динамических библиотеках. В таком коде используются не абсолютные (фиксированные) виртуальные позиции (адреса), а относительные (плавающие), что позволяет корректно использовать динамическую библиотеку, размещенную практически с любого виртуального адреса.

Опция `-ldl` означает компоновку с библиотекой `libdl`.

Опция `-f` (для команды `rm`) означает, что не будет запрашиваться подтверждений, будет удаляться все, что возможно.

Опция `-o` создает выходной исполняемый файл (т.е. создает и объектные файлы и их сборку в бинарный файл).

Опция `-c` создает объектные файлы, не собирая их в исполняемый файл (т.е. только компиляция, без линкования).

Опции `-Wall` `-Wextra` позволяет компилятору выводить все предупреждения. Эту опцию целесообразно использовать всегда, чтобы генерировать наиболее безошибочный код.

Задание 6 по предмету Операционные Системы: Создание статических и динамических библиотек в Linux (4846d.ru)