Операционные системы и системное программное обеспечение

Лабораторная работа №7

Целью работы является знакомство с механизмами, позволяющими использовать и создавать статические и совместно используемые библиотеки.

1. БИБЛИОТЕКИ И ЗАГОЛОВОЧНЫЕ ФАЙЛЫ

Библиотеки позволяют разным программам использовать один и тот же объектный код. Это избавляет программиста от необходимости создавать то, что уже создано.

Библиотека (library) — это набор соединенных определенным образом объектных файлов. Библиотеки подключаются к программе на стадии компоновки. Функция printf() , например, 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.

Библиотеки подразделяются на две категории:

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

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

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.

4.2. Подключение библиотек

Рассмотрим в качестве примера программу для возведения числа в степень (листинг 1).

```
#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;
}
```

Листинг 1. Программа power.c

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.

3. СОЗДАНИЕ СТАТИЧЕСКИХ БИБЛИОТЕК

Статическая библиотека — это архив, создаваемый специальным архиватором ar из пакета GNU binutils.

Архиватор объединяет несколько файлов в один с возможностью выполнения обратной процедуры.

Утилита ar создает архив, который может подключаться к программе во время компоновки на правах библиотеки. Этот архиватор поддерживает много опций, однако нас интересуют только некоторые из них.

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 теперь пример создания статической библиотеки, в которой реализуются две функции для работы с окружением. Для наглядности разобьем исходный код на два файла: mysetenv.c (листинг 2) и myprintenv.c (листинг 3).

```
#include <stdlib.h>
#include <stdio.h>
#include "myenv.h"

void mysetenv (const char * name, const char * value)
{
    printf ("Setting variable %s\n", name);
    setenv (name, value, 1);
}
```

Листинг 2. Файл mysetenv.c

```
#include <stdlib.h>
#include <stdio.h>
#include "myenv.h"

void myprintenv (const char * name)
{
    char * value = getenv (name);
    if (value == NULL) {
        printf ("Variable %s doesn't exist\n");
        return;
    }
    printf ("%s=%s\n", name, value);
}
```

Листинг 3. Файл myprintenv.c

Чтобы функции mysetenv() и myprintenv() вызывались по правилам, нужно создать заголовочный файл (листинг 4).

```
#ifndef MYENV_H
#define MYENV_H
void mysetenv (const char * name, const char * value);
void myprintenv (const char * name);
#endif
```

Листинг 4. Заголовочный файл myenv.h

Обратите внимание на следующие строки листинга 4:

```
#ifndef MYENV_H
#define MYENV_H
#endif
```

Этот распространенный "фокус" с препроцессором позволяет избежать множественных включений в исходный код одного и того же заголовочного файла.

Теперь организуем автоматическую сборку библиотеки. Для этого необходимо создать make-файл (листинг 5).

```
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
```

Листинг 5. Файл Makefile

Напишем программу, к которой будет подключаться полученная библиотека (листинг 6).

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

int main (int argc, char ** argv)
{
    if (argc < 3) {
        fprintf (stderr, "Too few arguments\n");
        return 1;
    }
    mysetenv (argv[1], argv[2]);
    myprintenv (argv[1]);

    return 0;
}
```

Листинг 6. Программа envmain.c

Осталось только собрать эту программу с библиотекой libmyenv.a:

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

Проверим, что получилось:

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

Кроме того, сборку программы и создание библиотеки можно объединить в один проект. Для этого создадим универсальный make-файл (листинг 7).

```
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
```

Листинг 7. Универсальный файл Makefile

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
```

4. 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 следует учитывать два нюанса:

- в процессе компоновки совместно используемой библиотеки должны участвовать объектные файлы, содержащие позиционно-независимый код (Position Independent Code). Этот 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 ;

- опция -L , указанная при компоновке, позволяет дополнить список каталогов, в которых будет выполняться 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.

В момент запуска программы для поиска библиотек просматриваются каталоги, перечисленные в файле /etc/ld.so.conf и в переменной окружения LD_LIBRARY_PATH .

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 разде- ленные 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.

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.

Теперь, уяснив все тонкости, переходим к делу. За основу возьмем пример из предыдущего раздела. Рассмотрим сначала концепцию использования переменной окружения LD_LIBRARY_PATH . Чтобы переделать предыдущий пример для работы с динамической библиотекой, требуется лишь изменить make-файл (листинг 8).

```
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
```

Листинг 8. Модифицированный файл Makefile

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.

5. 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 С:

```
$ 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.

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), что иллюстрируют листинги 9 – 12.

```
#include <stdio.h>
#include "common.h"
void do_first (void)
{
printf ("First library\n");
}
```

Листинг 9. Исходный файл first.c

```
#include <stdio.h>
#include "common.h"
void do_second (void)
{
printf ("Second library\n");
```

```
}
```

Листинг 10. Исходный файл second.c

```
#include "common.h"
void do_third (void)
{
do_first ();
do_second ();
}
```

Листинг 11. Исходный файл third.c

```
#include "common.h"
int main (void)
{
do_third ();
return 0;
}
```

Листинг 12. Программа program.c

Общий для всех файл common.h содержит объявления библиотечных функций (листинг 13).

```
#ifndef COMMON_H
#define COMMON_H
void do_first (void);
void do_second (void);
void do_third (void);
#endif
```

Листинг 13. Общий файл common.h

Теперь создадим make-файл, который откомпилирует все исходные файлы и создаст три библиотеки, одна из которых будет статической (листинг 14).

```
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
```

Листинг 14. Файл Makefile

Итак, make-файл содержит семь целевых связок. Рассмотрим каждую из них по порядку.

1. Бинарник program — создается в один прием из исходного файла program.c с подключением динамической библиотеки libthird.so. Символ \ (бэкслэш) применяется в make-файлах для разбиения длинных строк.
2. Динамическая библиотека libthird.so — получается в результате компоновки файла third.o с подключением динамической библиотеки libfirst.so и архива libsecond.a.
3. Объектный файл third.o, содержащий позиционно-независимый код, — создается компиляцией исходного файла third.c с опцией -fPIC .
4. Совместно используемая библиотека libfirst.so — создается из исходного файла first.c. Компиляция и компоновка полученного позиционно-независимого объектного кода осуществляются в один прием.
5. Статическая библиотека libsecond.a — создается путем архивирования единственного файла second.o.
6. Файл second.o — получается в результате компиляции исходника second.c. Как уже отмечалось, для создания статических библиотек используются файлы, содержащие обычный объектный код с фиксированными позициями.
7. Целевая связка clean — очищает проект, удаляя исполняемую программу, библиотеки и объектные файлы.

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.

ЗАДАНИЕ

Написать и протестировать работу функции из лабораторной работы 6, создав и использовав:

- статическую библиотеку;
- динамическую библиотеку.

Создать *make*-файл(ы) для компиляции и сборки библиотек.