Министерство науки и высшего образования Российской Федерации
Санкт-Петербургский Политехнический Университет Петра Великого
—
Институт прикладной математики и механики
Высшая школа кибербезопасности и защиты информации

ЛАБОРАТОРНАЯ РАБОТА № 1

«**Компилятор, компоновщик и динамические библиотеки**»

по дисциплине «Технологии разработки современного программного обеспечения»

Выполнил
студент гр. 3651003/70801 Гасанов Э.А.

<подпись>

Преподаватель
доцент Никольский А.В.

<подпись>

Санкт-Петербург
2020

Формулировка задания

Написать программу, использующую три библиотеки: zlib, libpng, freetype. Программа должна осуществлять создание png файла с нарисованным в нем текстовым сообщением с помощью шрифта, загруженного из файла *.ttf.

При написании программы должны быть выполнены несколько условий:

1. Программа должна компилироваться с помощью gcc и системы makefile в ОС Linux или WSL.
2. Программа должна быть выполнена в двух версиях:
    a. static - статическая компоновка всех трех библиотек.
    b. dynamic – динамическая загрузка всех трех библиотек в начале работы с помощью dlload.
    c. blob – реализация всего функционала в виде блоба и загрузка его с диска с помощью загрузчика elf-loader, который должен быть включен в состав проекта программы.
3. Система Makefile должна поддерживать следующие команды:
    a. make static – компиляция исходников в версии со статической компоновкой;
    b. make dynamic – компиляция исходников в версии с динамической компоновкой;
    c. make blob – компиляция исходников в версии с компоновкой в виде блоба;
    d. make clean – удаление всех бинарных файлов;
    e. make all – удаление старых бинарных файлов и компиляция исходников в версиях static, dynamic и blob;
4. Программа в независимости от типа сборки по результатам своей работы должна не только генерировать png файл с нарисованным текстовым сообщением в нем, но и печатать на экран общее время своей работы и время, затраченное на загрузку системы (от начала main до конца своей работы).
5. Файл с шрифтом, имя png файла и печатаемый текст должны поступать в программу из аргументов командной строки.
6. Программа не должна генерировать исключения или преждевременно завершать свою работу в независимости от аргументов командной строки.

Ход работы

Для данной работы были скачены необходимые файлы трёх библиотек: zlib, libpng, freetype. Затем была произведена их сборка: статически и динамически, для чего использовались специально написанные make-файлы.

Makefile – файл, содержащий набор инструкций для программы make. Программа make с помощью этого файла позволяет автоматизировать процесс компиляции программы и выполнять при этом различные действия.

Итак, в результате сборки были получены статические библиотеки. В системах UNIX командой для сборки статичной библиотеки обычно является ar, и библиотечный файл, который при этом получается, имеет расширение *.a. Также эти файлы обычно имеют префикс «lib» в своём названии, и они передаются компоновщику с опцией "-l" с последующим именем библиотеки без префикса и расширения. Всякий раз, когда файлы добавляются в библиотеку, включая первоначальное создание библиотеки, библиотека должна быть проиндексирована, что делается с помощью команды ranlib. Ranlib создает в библиотеке заголовок с символами содержимого объектного файла. Это помогает компилятору быстрее ссылаться на символы. Большая библиотека может иметь тысячи символов, что означает, что индекс может значительно ускорить поиск ссылок. При окончательной сборке, то есть и компиляции, и компоновке мы используем -I и путь к хэдерам, а так же -L, которая укажет путь к библиотеке. То есть окончательная команда принимает вид:

gcc app/app-static.c -L freetype/ -l freetype -L libpng/ -l png -L zlib/ -l zlib -lm -I ./libpng/ -I ./freetype/ -std=gnu99 -o app-static.out

Алгоритм работы по созданию текста в картинке примитивен: принимаются аргументы командной строки, происходит их обработка в функции parse_arguments. Затем происходит рендер глифа, а после – генерация png-картинки.

Динамические библиотеки в системах UNIX имеют расширение .so. Они подключаются к программе в момент выполнения. Если компоновщик обнаруживает, что определение конкретного символа находится в разделяемой библиотеке, то он не включает это определение в конечный исполняемый файл. Вместо этого компоновщик записывает имя символа и библиотеки, откуда этот символ должен предположительно появиться.

И более того, если конкретный символ берётся из конкретной динамической библиотеки (скажем printf из libc.so), то всё содержимое библиотеки помещается в адресное пространство программы. Это основное отличие от статических библиотек, где добавляются только конкретные объекты, относящиеся к неопределённому символу.

Для сборки динамических библиотек используется аргумент –fPIC: генерируется «переносимый код», то есть все абсолютные адреса будут заменяться на относительные.

Чтобы осуществить динамическую сборку мы воспользуемся уже известыми -I и -L, а так же вместо LD_LIBRARY_PATH, воспользуемся rpath по причинам:

- Безопасность: каталоги, указанные в LD_LIBRARY_PATH, ищутся до стандартных местоположений. Таким образом, злоумышленник может заставить приложение загрузить версию общей библиотеки, которая содержит вредоносный код!
- Производительность: link loader должен искать все указанные каталоги, пока не найдет каталог, в котором находится общая библиотека - для ВСЕХ общих библиотек, с которыми связано приложение! Это означает, что многие системные вызовы open () могут завершиться с ошибкой «ENOENT (нет такого файла или каталога)»! Если путь содержит много каталогов, количество неудачных вызовов будет линейно увеличиваться, и мы можем понять это по времени запуска приложения.
- Несоответствие: это самая распространенная проблема. LD_LIBRARY_PATH заставляет приложение загружать разделяемую

библиотеку, с которой она не была связана, и это, скорее всего, несовместимо с исходной версией. В результате команда имеет вид:

cd app && gcc app-dynamic.c -Wl,-rpath,/freetype,-rpath,/libpng,-rpath,/zlib -L../freetype -lfreetype -I ../freetype -L../libpng -lpng -L../zlib/zlib.so -I ../zlib/ -ldl -o ../app-dynamic.out

Алгоритм работы app-dynamic сводится к следующему: разбор аргументов командной строки, затем загрузка библиотек с помощью dl-функций, рендер глифа, рендер картинки.

Для сборки типа blob, мы воспользовались elf-загрузчиком, который загружает .so-файл в текущее адресное пространство и запускает функцию. То есть binary blob – это .so, динамически линкуемый в runtime. Алгоритм сборки выглядит следующим образом:

1.Необходимо собрать статическую библиотеку из app-static.c

```
elvin@ubuntu:~/lab1/app$ gcc -c -fPIC app-static.c -I ../freetype/ -I ../libpng/ -I ../zlib/
```

2.Затем используем ranlib

```
elvin@ubuntu:~/lab1KR/app$ ar -rsc libblob.a *.o
```

3. Собираем все статические библиотеки(zlib,freetype,libpng, а теперь ещё и libblob) в одну динамическую:

```
elvin@ubuntu:~/lab1KR/app$ gcc -shared -fPIC -o c.so -Wl,--whole-archive libblob.a ../freetype/libfreetype.a ../libpng/libpng.a ../zlib/libzlib.a -Wl,--no-whole-archive
```

4. Соберем всё:

```
elvin@ubuntu:~/lab1KR/app$ gcc -Wall -g -o elfloader elf_loader.c main.c wheelc/list.c -ldl
```

Так получили сборку blob.

Структура каталогов

a. lab1\ - каталог со всеми исходными текстами и скриптами сборки
b. lab1\zlib\ - исходные тексты и makefile для библиотеки zlib
c. lab1\freetype\ - исходные тексты и makefile для библиотеки freetype
d. lab1\libpng\ - исходные тексты и makefile для библиотеки libpng
e. lab1\app\ - исходные тексты и wheels для blob
f. lab1\makefile – корневой makefile

Сравнительная таблица о плюсах и минусах статических и динамических библиотек

Статические библиотеки		Динамические библиотеки	
+	-	+	-
Статическая библиотека позволяет разделять код, используя повторно объектный файл. То есть она собирается вовремя компиляции программы, дублируя свой код в исходный код программы. Экономия времени перекомиплирования.	Время перекопилировки стало менее важным фактором из-за появления быстрых современных компиляторов. А также, если код библиотеки обновлён, то нужно перекопилироваться, так как код физически находится внутри исполняемого файла. Каждой программе, которая	Динамическое связывание не требует копирования кода. Фактическое связывание происходит при запуске программы, когда исполняемый файл и библиотека находятся в памяти. Если конкретный символ берётся из конкретной динамической библиотеки (скажем printf из libc.so), то всё содержимое библиотеки помещается в адресное	Требуется больше времени на загрузку и запуск программы. Не может иметь полный функционал в отсутствие библиотеки.

	использует эту библиотеку, физически помещается объектный код из библиотеки в код исполняемый файл. А значит исполняемые файлы занимают больше памяти. А также проблема циклических зависимостей.	пространство программы. Экономия памяти.	
Статические библиотеки часто полезны для разработчиков, если они хотят разрешить программистам ссылаться на свою библиотеку, но не хотят предоставлять исходный код библиотеки	Не является преимуществом для программиста, пытающегося использовать библиотеку	Если происходит какое-либо обновление функции внутри библиотеки, то не нужно перекопилировать свою программу. Обновится только библиотека, так как загружается разделяемая библиотека по ранодомизированному адресу(обычно он разный).	Уязвимый ldd. Например, ldd/bin/grep: Ldd -это обёртка над **LD_TRACE_LOADED_OBJECTS = 1 / bin / grep** и **LD_TRACE_LOADED_OBJECTS = 1 /lib/ld-linux.so.2** Они устанавливают переменную среды LD_TRACE_LOADED_OBJECTS. /bin/grep никогда не запускался. Это особенность динамического загрузчика GNU. Если он замечает переменную среды LD_TRACE_LOADED_OBJECTS, он никогда не выполняет программу, он выводит список зависимостей динамической библиотеки и завершает работу.

Теоретически, код в статических библиотеках ELF, который связан с исполняемым файлом, должен работать немного быстрее (на 1-5%), чем разделяемая библиотека или динамически загружаемая библиотека	На практике это редко случается из-за мешающих факторов		В linux 1-я команда выполняется, если указанная программа ldd не может быть загружена ld-linux.so загрузчиком, и что 2-я команда выполняется, если она может. Один конкретный случай, когда программа не будет обрабатываться ld-linux.so- это когда у нее загрузчик, отличный от системного по умолчанию, указанного в разделе ELF .interp. В этом и заключается идея выполнения произвольного кода с помощью ldd загрузки исполняемого файла через другой загрузчик, который не обрабатывает переменную среды LD_TRACE_LOADED_OBJECTS, а выполняет программу.

Основываясь на основную разницу между динамическими и статическими библиотеками, если нас волнует размер(в меньшую сторону) исполняемого бинарника, то лучше использовать разделяемые библиотеки, так как в итоге это ещё и повлияет на масштабируемость приложения в лучшую сторону. Для небольших проектов, где размер не критичен – статические.

Код makefile

Корневой makefile

```
CC=gcc
L1=freetype
L2=libpng
L3=zlib
STATIC=app/app-static.c
DYNAMIC=app/app-dynamic.c
STATIC_OUT=app-static.out
DYNAMIC_OUT=app-dynamic.out
all: clean static dynamic blob
rebuild:
	@cd $(L1)/ && make all
	@cd $(L2)/ && make all
	@cd $(L3)/ && make all

static:
	@$(CC) $(STATIC) -L $(L1)/ -l $(L1) -L $(L2)/ -l png -L $(L3)/ -l zlib -lm -I ./$(L2)/ -I ./$(L1)/ -std=gnu99 -o $(STATIC_OUT)
dynamic:
	cd app && gcc app-dynamic.c -Wl,-rpath,/freetype,-rpath,/libpng,-rpath,/zlib -L../freetype -lfreetype -I ../freetype  -L../libpng -lpng -L../zlib/zlib.so  -I ../zlib/ -ldl -o ../app-dynamic.out

#@$(CC) -I ./$(L1)/ -I ./$(L2)/ -std=gnu99 -rdynamic $(DYNAMIC) -o $(DYNAMIC_OUT) -ldl
blob:
	@cd app && gcc -c -fPIC app-blob.c -I ../freetype/ -I ../libpng/ -I ../zlib/ -lm && ar -rsc libblob.a *.o &&  gcc -shared -fPIC -o c.so -Wl,--whole-archive libblob.a ../freetype/libfreetype.a ../libpng/libpng.a ../zlib/libzlib.a  -Wl,--no-whole-archive -
```

```
lm && gcc -Wall -g -o elfloader.out elf_loader.c main.c wheelc/list.c -ldl && mv c.so ../ && mv elfloader.out ../
clean:
	rm -f *.out
	rm -f *.png
	rm -f *.so
	rm -f *.txt
	@cd app &&  rm -f *.o && rm -f *.a
```

Makefile zlib

```
CC=gcc
LD=ld
CFLAGS=-c -Wall
LDFLAGS=
SOURCES = adler32.c compress.c crc32.c deflate.c gzclose.c gzlib.c gzread.c gzwrite.c infback.c inffast.c inflate.c inftrees.c trees.c uncompr.c zutil.c
OBJECTS=$(SOURCES:.c=.o)
DYNFLAGS=-c -shared -fPIC

all: clean static dynamic blob

dynamic: $(OBJECTS)
	$(LD) -shared -o "zlib.so" *.o -lc

blob: static

static: $(OBJECTS)
	ar rc libzlib.a $(OBJECTS)
	ranlib libzlib.a
```

```
.c.o: $(SOURCES)
	$(CC) $(DYNFLAGS) $< -o $@

clean:
	rm -f *.o
	rm -f *.a
	rm -f *.so
```

Makefile libpng

```
CC=gcc
LD=ld
CFLAGS=-c -Wall
LDFLAGS=
SOURCES = png.c pngerror.c pngget.c pngmem.c pngpread.c pngrio.c pngrtran.c
pngrutil.c pngset.c pngtrans.c pngwio.c pngwrite.c pngwtran.c pngwutil.c
OBJECTS=$(SOURCES:.c=.o)
DYNFLAGS=-c -shared -fPIC -lc
INCLUDE = ../zlib

all: clean static dynamic blob

dynamic: $(OBJECTS)
	$(LD) -shared -o "libpng.so" *.o

blob:static

static: $(OBJECTS)
	ar rc libpng.a $(OBJECTS)
	ranlib libpng.a
```

```
.c.o: $(SOURCES)
	$(CC) $(DYNFLAGS) -I$(INCLUDE) $< -o $@

clean:
	rm -f *.o
	rm -f *.a
	rm -f *.so
```

Makefile freetype

```
CC=gcc
LD=ld
CFLAGS=-c -Wall
LDFLAGS=
SOURCES = \
	autofit.c \
	bdf.c \
	cff.c \
	pfr.c \
	pcf.c \
	sfnt.c \
	raster.c \
	pshinter.c \
	winfnt.c \
	truetype.c \
	type1.c \
	type42.c \
	psaux.c \
	psnames.c \
	type1cid.c \
```

```
	ftbase.c \
	ftbitmap.c \
	ftcache.c \
	ftdebug.c \
	ftgasp.c \
	ftglyph.c \
	ftgzip.c \
	ftinit.c \
	ftlzw.c \
	ftstroke.c \
	ftsystem.c \
	smooth.c \

OBJECTS=$(SOURCES:.c=.o)
DYNFLAGS=-c -shared -fPIC -lc

all: clean static dynamic blob

dynamic: $(OBJECTS)
	$(LD) -shared -o "freetype.so" *.o

blob: static

static: $(OBJECTS)
	ar rc libfreetype.a $(OBJECTS)
	ranlib libfreetype.a

.c.o: $(SOURCES)
	$(CC) $(DYNFLAGS) -I./ $< -o $@
```

```
clean:
	rm -f *.o
	rm -f *.a
	rm -f *.so
```