Создание динамических библиотек при помощи набора компиляторов и утилит GCC и их применение.

Цель работы.

Изучить процесс создания динамических библиотек при помощи набора компиляторов и утилит GCC и особенности их применения.

Динамическая библиотека - это созданная специальным образом библиотека, которая присоединяется к результирующей программе в два этапа. Первый этап, это 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 осуществляет системный динамический загрузчик во время запуска программы. Загрузчик проверяет все библиотеки связанные с программой 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 при этом если другая запускаемая программа связана с этой же загруженной динамической библиотекой, то она использует туже 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 подобных библиотек имеет свои особенности в разных операционных системах, которые будут обозначены далее. При выполнении задания, допускается использовать любой способ, но при защите лабораторной необходимо продемонстрировать знание обоих.

Порядок выполнения работы.

Пример создания динамической библиотеки.

Для примера создадим следующие файлы:

//----------------------fun.h

```
void func(void);
```

//----------------------first.c

```
#include <stdio.h>
#include "fun.h"
void func(void)
{
        printf("First function...\n");
}
```

```
//----------------------second.c
#include <stdio.h>
#include "fun.h"
void func(void)
{
        printf("Second function...\n");
}

//----------------------main.c

#include <stdio.h>
#include "fun.h"
int main(void)
{
        func();
        printf("Main function...\n");
        return 0;
}
```

Файлы *.c необходимо скомпилировать в объектные файлы, при этом рекомендуется использовать параметр **-fPIC**

Опция **-fPIC** - требует от компилятора, при создании объектных файлов, порождать позиционно-независимый код (PIC - Position Independent Code), его основное отличие в способе представления адресов. Вместо указания фиксированных (статических) позиций, все адреса вычисляются исходя из смещений заданных в глобальной таблицы смещений (global offset table - GOT). Формат позиционно-независимого кода позволяет подключать исполняемые модули к коду основной программы в момент её загрузки. Соответственно, основное назначение позиционно-независимого кода - создание динамических (разделяемых) библиотек.

Для ОС семейства Windows в простейшем случае создание динамических библиотек выглядит так:

gcc -shared -o lib1.dll first.o

gcc -shared -o lib2.dll second.o

Опция **-shared** - указывает gcc, что в результате должен быть собран не исполняемый файл, а разделяемый объект - динамическая библиотека.

А их подключение - так:

gcc -o main1.exe main.o -L./ -llib1

gcc -o main2.exe main.o -L./ -llib2

Также создадим статическую библиотеку с первым объектным файлом:

ar.exe crs first.lib first.o

и сравним размеры статической и динамической библиотек. Как видно, размер динамической значительно больше.

Подключение динамических библиотек в процессе выполнения программы.

Для начала создадим библиотеку с функциями для загрузки других библиотек.

```
    //----------------------load.c
#include "load.h"
#include "fun.h"
#include <stdio.h>
#include <windows.h>

void LoadRun(const char * const s) {
   void * lib;
   void (*fun)(void);
   lib = LoadLibrary(s);
   if (!lib) {
     printf("cannot open library '%s'\n", s);
     return;
   }
   fun = (void (*)(void))GetProcAddress((HINSTANCE)lib, "func");
   if (fun == NULL) {
     printf("cannot load function func\n");
   } else {
     fun();
   }
   FreeLibrary((HINSTANCE)lib);
}
```

Здесь стоит обратить внимание на функции **LoadLibrary** и **GetProcAddress**. Первой мы загрузим в память библиотеку, переданную ей в качестве параметра. Вторая используется для получения указателя на функцию из загруженной библиотеки с указанным названием. В конце работы библиотеку необходимо выгрузить функцией **FreeLibrary**.

В данном примере мы ищем в библиотеке функцию “**func**”.

```
    //---------------------- load.h
void LoadRun(const char * const s);
```

Создадим динамическую библиотеку из load.c:

gcc.exe -c -fPIC load.c

gcc.exe -shared -o load.dll load.o

Теперь опишем файл основной программы. В качестве подключаемых библиотек будем использовать ранее созданные **lib1.dll** и **lib2.dll**.

```
    //---------------------- loadmain.c
#include "load.h"
#include <stdio.h>
```

```
int main(void)
{
  int a=0,b=1;
  printf("Choose library:\n1-first.\n2-second,\n3-quit\n");
  while(b)
  {
        scanf("%d",&a);
        if(a==1)
           LoadRun("lib1.dll");
        if(a==2)
           LoadRun("lib2.dll");
        if(a==3)
           b=0;
  }
  return 0;
}
```

Скомпилируем и запустим полученную программу.
gcc.exe -c -fPIC loadmain.c
gcc.exe -o loadmain.exe loadmain.o -L./ -lload
loadmain.exe

Теперь при нажатии на 1 у нас будет вызываться функция из первой библиотеке, при нажатии на 2 - из второй.

Особенности создания и использования динамических библиотек в ОС семейства Linux.

В случае использовании ОС семейства Linux, необходимо учитывать следующие особенности:

- вместо **windows.h** в загрузчике следует использовать заголовочный файл **dlfcn.h**, , загрузку библиотеки следует выполнять так: **lib = dlopen(s, RTLD_LAZY)**;
- получение функции из библиотеки происходит так: **fun = (void (*)(void))dlsym(lib, "run")**, освобождение библиотеки **- dlclose(lib)**;
- при компиляции основного файла необходимо в конец команды добавить параметр **–ldl** (подключить стандартную библиотеку **dl**);
- для динамических библиотек принято использовать расширение ***.so**

Контроль версий.

При разработке проекта с использованием динамических библиотек необходимо тщательно контролировать версию библиотеки во избежание проблем совместимости. В самом проекте этот

контроль осуществляется различными способами в зависимости от семейства ОС. Рассмотрим эти способы.

Для ОС семейства Windows контроль версии осуществляется путём простого добавления информации о версии в имя файла и проверки его при компиляции проекта.

Для ОС семейства Linux введено понятие soname. Продемонстрируем его применение на примере. Для начала скомпилируем библиотеку с указанием soname:

gcc -shared -o libhello.so.2.4.0.5 -Wl,-soname,libhello.so.2 first.o second.o

Результатом будет файл **libhello.so.2.4.0.5**, где 2.4.0.5 - его версия, а liibhello.so - имя. Для задания soname используется параметр **-Wl,-soname,libhello.so.2** Данный параметр предназначен для непосредственного взаимодействия пользователя с линковщиком. По ходу компиляции gcc вызывает линковщик автоматически, автоматически же, по собственному усмотрению, gcc передает ему необходимые для успешного завершения задания параметры. Если у пользователя возникает потребность самому вмешаться в процесс линковки, он может воспользоваться специальным параметром **gcc -Wl,-option,value1,value2....** Что означает передать линковщику (**-Wl**) параметр **-option** с аргументами **value1**, **value2** и так далее. В нашем случае линковщику был передан параметр **-soname** с аргументом **libhello.so.2.**

Для того чтобы система, конкретно загрузчик динамических библиотек, имела представление о том, библиотека какой версии была использована при компиляции приложения и, соответственно, необходима для его успешного функционирования, был предусмотрен специальный идентификатор **- soname**, помещаемый как в файл самой библиотеки, так и в исполняемый файл приложения. Идентификатор soname это строка, включающая имя библиотеки с префиксом **lib**, точку, расширение **so**, снова точку и оду или две (разделенные точкой) цифры версии библиотеки **- libname.so.x.y**. То есть soname совпадает с именем файла библиотеки вплоть до первой или второй цифры номера версии. Пусть имя исполняемого файла нашей библиотеки **libhello.so.2.4.0.5**, тогда soname библиотеки может быть **libhello.so.2**. При изменении интерфейса библиотеки её soname необходимо изменять! Любая модификация кода, приводящая к несовместимости с предыдущими релизами, должна сопровождаться появлением нового soname.

Приведём пример. Пусть для успешного исполнения некоторого приложения необходима библиотека с именем **hello**, пусть в системе таковая имеется, при этом имя файла библиотеки **libhello.so.2.4.0.5,** а прописанное в нем soname библиотеки **libhello.so.2**. На этапе компиляции приложения, линковщик, в соответствии с опцией **-lhello**, будет искать в системе файл с именем **libhello.so**. В реальной системе **libhello.so** это символическая ссылка на файл **libhello.so.2.4.0.5**. Получив доступ к файлу библиотеки, линковщик считает прописанное в нем значение soname и наряду с прочим поместит его в исполняемый файл приложения. Когда приложение будет запущено, загрузчик динамических библиотек получит запрос на подключение библиотеки с soname, считанным из исполняемого файла, и попытается найти в системе библиотеку, имя файла которой совпадает с soname. То есть загрузчик попытается отыскать файл **libhello.so.2.** Если система настроена корректно, в ней должна присутствовать символическая ссылка **libhello.so.2** на файл **libhello.so.2.4.0.5**, загрузчик получит доступ к требуемой библиотеки и далее, не задумываясь (и ни чего более не проверяя) подключит её к приложению. Теперь представим, что мы перенесли откомпилированное таким образом приложение в другую систему, где развернута только предыдущая версия библиотеки с soname **libhello.so.1**. Попытка запустить программу приведет к ошибке, так как в этой системе файла с именем **libhello.so.2** нет.

В виду этого, в ОС семейства linux недостаточно просто скомпилировать программу командой **gcc -o main main.o -L. -lhello -Wl,-rpath**,. Сначала необходимо создать символическую ссылку на библиотеку (в нашем случае командой **ln -s libhello.so.2.4.0.5 libhello.so)** и другую, соответствующую soname, командой **ln -s libhello.so.2.4.0.5 libhello.so.2**

Также командой **objdump -p main | grep NEEDED** мы всегда можем проверить библиотеки с каким soname требуются указанной программе.

Также при компиляции использовался параметр **-Wl,-rpath,**. - передать линковщику параметр **-rpath** с аргументом .. С помощью **-rpath** в исполняемый файл программы можно прописать дополнительные пути по которым загрузчик разделяемых библиотек будет производить поиск библиотечных файлов. В нашем случае прописан путь . - поиск файлов библиотек будет начинаться с текущего каталога.

Узнать какие разделяемые библиотеки необходимы приложению можно и с помощью утилиты **ldd: ldd main**

Использование условной директивы препроцессора.

Иногда требуется чтобы написанная программа могла компилироваться и работать на операционных системах из разных семейств. Для этого можно использовать директивы препроцессора. Подробно о них можно почитать тут:
https://msdn.microsoft.com/ru-ru/library/3sxhs2ty(v=vs.120).aspx

Сейчас же отметим только директиву **#ifdef ИМЯ**. Данная директива проверяет определен ли указанный идентификатор. Значение идентификатора при этом не важно, важен сам факт определения. Вместо неё также можно использовать директиву **#if defined(ИМЯ)**. Если указанный идентификатор не определён, то блок текста до директивы **#endif** будет проигнорирован. Также существуют директивы #else аналогичная по принципу работы оператору else, и директива **#ifndef**, обратная директиве **#ifdef**. Пример использования:

```
#ifndef WIN32
ДЕЙСТВИЕ1
#else
ДЕЙСТВИЕ2
#endif
```

В данном примере **ДЕЙСТВИЕ1** выполняется только на ОС отличных от ОС семейства Windows, в остальных случаях выполняется **ДЕЙСТВИЕ2**.

Использование динамических библиотек, написанных на разных языках.

Одной из особенностей динамических библиотек является возможность подключения к программе, написанной на одном языке библиотеки, написанной на другом. Продемонстрируем это.

Изменим файл loadmain.c:

```
    //---------------------- loadmain.c
#include "load.h"
#include <stdio.h>
```

```
int main(void)
{
  int a=0,b=1;
  printf("Choose library:\n1-first.\n2-second,\n3-pascal\n4-quit\n");
  while(b)
  {
        scanf("%d",&a);
        if(a==1)
           LoadRun("lib.dll");
        if(a==2)
           LoadRun("lib2.dll");
        if(a==3)
           LoadRun("funcmod.dll");
        if(a==4)
           b=0;

  }
  return 0;
}
```

Теперь напишем библиотеку, используя в качестве языка программирования Pascal:

```
    //-----func.pas
unit func;

interface
  procedure func(); stdcall; external name 'func';
implementation
end.

//----funcmod.pas
library modul;
uses
  func;

procedure func; stdcall;
begin
  write('Pascal function...');
end;

exports
  func;
```

```
begin
end.
```

Теперь скомпилируем библиотеку и основную программу, в качестве компилятора pascal использовать будем Free Pascal.

fp -Cp .\funcmod.pas
gcc.exe -c -fPIC .\Loadmain.c
gcc.exe -o loadmain.exe loadmain.o -L./ -lload

Теперь запустив программу мы при нажатии на 3 получим текст функции из новой библиотеки.

Задания к лабораторной работе.

1. Написать программу в соответствие с вариантом.
2. Массив и матрицу заполнять случайными числами от -50 до 50.
3. Функции для работы с массивами и матрицами поместить в две отдельные динамические библиотеки.
4. При запуске программы пользователю должно быть представлено меню, в котором можно выбрать с чем будет происходить работа: с матрицей или с массивом.
5. В зависимости от выбора пользователя, загружается одна или другая динамическая библиотека.
6. Библиотеки должны быть скомпилированы с учётом возможного использования в ОС семейств Linux или Windows/
7. Основная программа должна при помощи директив препроцессора поддерживать мультиплатформенность в рамках этих двух семейств ОС.

Варианты к заданию.

Вариант 1

Найти сумму наибольших из отрицательных элементов матрицы A (7x8) и массива B (76)

Вариант 2

В матрице A (7x6) и массиве B (67) заменить все отрицательные числа их квадратами.

Вариант 3

Отсортировать массив M(50) и строки матрицы A(6x7) в порядке

убывания

Вариант 4

Подсчитать общее количество отрицательных элементов в массиве D(42) и матрице A(6x9).

Вариант 5

В массиве A (76) и матрице B (5x6) заменить все числа, кратные 3, единицами.

Вариант 6

Подсчитать число элементов матрицы Q (9x11) и массива R (38), остаток от деления которых на пять равен единице.

Вариант 7

Подсчитать число элементов матрицы Q (5x7) и массива R(57), кратных трем.

Вариант 8

Определить количество элементов массива A (35) и матрицы D(5x7), кратных 3 и не кратных 5 одновременно.

Вариант 9

Вычислить сумму элементов массива M (11) и матрицы S (4x11), значения которых лежат в введенном с клавиатуры диапазоне [X, Y].

Вариант 10

Заменить все четные по значению элементы массива A (20) и матрицы B (5x6) на их квадраты,

Вариант 11

Найти наибольший отрицательный элемент массива A (23) и матрицы B (7x5).

Вариант 12

В массиве A (45) и каждой строке матрицы B (10x6) найти локальные максимумы, определить их местоположение. Локальным максимумом назовем элемент массива, значение которого больше, чем значения двух соседних с ним элементов.

Вариант 13

В массиве A (35) и матрице B (8x5) найти минимум, определить его местоположение (с учетом возможного повторения).

Вариант 14

В массиве K (73) и матрице R (7x10) найти число элементов, которые делятся на 7 без остатка.

Вариант 15

В массиве A(35) и в каждой строке матрицы B (9x8) сменить знак максимального по модулю элемента на противоположный с учётом повторений.

Контрольные вопросы.

1) Отличия динамических библиотек от статических.
2) Загрузка динамических библиотек во время выполнения программы.
3) Отличия при создании и использовании динамических библиотек в ОС семейств Linux и Windows.
4) Контроль версий динамических библиотек.
5) Препроцессор.

Использована статья “О GCC, компиляции и библиотеках“, расположенная по адресу http://pyviy.blogspot.ru/2010/12/gcc.html

Рекомендованы к чтению:
https://cygwin.com/cygwin-ug-net/dll.html

http://www.transmissionzero.co.uk/computing/building-dlls-with-mingw/
https://msdn.microsoft.com/ru-ru/library/3sxhs2ty(v=vs.120).aspx

Задачи взяты из методического пособия “Основы программирования на языке Си”, составители:
Арипова Ольга Владимировна, Бузюкина Ольга Александровна