Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет: “Информационные технологии и прикладная математика”
Кафедра: 806 “Вычислительная математика и программирование”

Лабораторная работа №5
по курсу “Операционные системы”

Студент	Полей-Добронравова А.В.
Группа	М8О-307Б-18
Вариант	7
Преподаватель	Миронов Е.С.
Дата	
Оценка	

Москва, 2020

1. Постановка задачи

Цель работы

Целью является приобретение практических навыков в:

• Создание динамических библиотек

• Создание программ, которые используют функции динамических библиотек

Задание

Требуется создать динамическую библиотеку, которая реализует определенный функционал. Далее использовать данную библиотеку 2-мя способами:

1. Во время компиляции (на этапе «линковки»/linking)
2. Во время исполнения программы, подгрузив библиотеку в память с помощью системных вызовов

В конечном итоге, программа должна состоять из следующих частей:

• Динамическая библиотека, реализующая заданных вариантом интерфейс;

• Тестовая программа, которая используют библиотеку, используя знания полученные на этапе компиляции;

• Тестовая программа, которая использует библиотеку, используя только местоположение динамической библиотеки и ее интерфейс. Провести анализ между обоими типами использования библиотеки.

7. Структура данных, с которой должна обеспечивать работу библиотека - массив. Тип данных, используемый структурой - целочисленный 32-битный.

2. Решение задачи

Создание динамически загружаемой библиотеки с разрешением .so:
gcc -c -fPIC arr.c
gcc -shared -o libmy_array.so arr.o

Ключ -shared указывает gcc на то, что следует создавать динамическую библиотеку, а ключ -fPIC даёт указание использовать при генерировании кода только команды с относительной адресацией "Generate position-independent code (PIC)".

Подключить динамическую библиотеку libmy_array.so на этапе линковки:
gcc -o run-stat main5static.o -L. -lmy_array -Wl,-rpath,.

Ключ -L. то, что искать её следует в текущем каталоге,
gcc -Wl,-option,value1,value2.... Что означает передать линковщику (-Wl) опцию -option с аргументами value1, value2 и так далее.
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.

Подгрузить динамическую библиотеку libmy_array.so с помощью **системных вызовов** из тела программы**:**

void *dlopen(const char ****filename**, int *flag*);**
dlopen загружает динамическую библиотеку, имя которой указано в строке *filename*, и возвращает прямой указатель на начало динамической библиотеки. Если *filename* указывает на NULL, то возвращается указатель на основную программу.
Значение *flag* должно быть одним из двух: RTLD_LAZY, подразумевающим разрешение неопределенных символов в виде кода, содержащегося в исполняемой динамической библиотеке; или RTLD_NOW, требующим разрешения всех неопределенных символов перед возвратом их из dlopen и возвращающим ошибку, если разрешение не может быть выполнено. Также значение RTLD_GLOBALможет быть задано через OR вместе с *flag;* в этом случае внешние символы, определенные в библиотеке, будут доступны загруженным позже библиотекам.

int dlclose(void *handle*);*

dlclose уменьшает на единицу счетчик ссылок на указатель динамической библиотеки *handle*. Если нет других загруженных библиотек, использующих ее символы и если счетчик ссылок принимает нулевое значение, то динамическая библиотека выгружается. Если динамическая библиотека экпортировала функцию, названную **_fini**, то эта функция вызывается перед выгрузкой библиотеки. **dlclose** возвращает 0 при удачном завершении и ненулевой результат при ошибке.

void *dlsym(void *_handle_, char *_symbol_);
dlsym использует указатель на динамическую библиотеку, возвращаемую dlopen, и оканчивающееся нулем символьное имя, а затем возвращает адрес, указывающий, откуда загружается этот символ. Если символ не найден, то возвращаемым значением **dlsym** является NULL; тем не менее, правильным способом проверки **dlsym** на наличие ошибок является сохранение в переменной результата выполнения **dlerror,** а затем проверка, равно ли это значение NULL. Это делается потому, что значение символа действительно может быть NULL.

Если по какой-либо причине выполнение **dlopen** неудачно, то она возвращает значение NULL. Понятный пользователю текст, описывающий большинство ошибок, происходящих при выполнении любых функций dl (dlopen, dlsym or dlclose), может быть получен при помощи функции **dlerror()**. **dlerror** возвращает NULL, если не возникло ошибок с момента инициализации или его последнего вызова. Если вызывать **dlerror()** дважды, то во второй раз результат выполнения всегда будет равен NULL.

Демонстрация работы программы:

```
MacBook-Pro-Amelia:Documents amelia$ make
gcc -c -fPIC -pthread -w -pipe -O2 -Wextra -Werror -Wall -Wno-sign-compare -pedantic -lm arr.c
gcc -pthread -w -pipe -O2 -Wextra -Werror -Wall -Wno-sign-compare -pedantic -lm -shared -o libmy_array.so arr.o
gcc -pthread -w -pipe -O2 -Wextra -Werror -Wall -Wno-sign-compare -pedantic -lm -o run-stat main5static.o -L. -lmy_array -Wl,-rpath,.
gcc -pthread -w -pipe -O2 -Wextra -Werror -Wall -Wno-sign-compare -pedantic -lm -o run-dyn main5.o -ldl
```

```
MacBook-Pro-Amelia:Documents amelia$ ./run-dyn
Massive created!
Massive:
 Elems:10
|233|1|2|3|4|5|6|7|8|9||
Massive:
 Elems:10
|233|1|2|3|4|5|6|7|8|9||
MacBook-Pro-Amelia:Documents amelia$ ./run-stat
Massive created!
Massive:
 Elems:10
|233|1|2|3|4|5|6|7|8|9||
Massive:
 Elems:10
|233|1|2|3|4|5|6|7|8|98||
```

3. Листинг программы

Makefile

```
CC = gcc

FLAGS = -pthread -w -pipe -O2 -Wextra -Werror -Wall -Wno-sign-compare -pedantic -lm

all: run

run: libmy_array.so main5.o main5static.o

	$(CC) $(FLAGS) -o run-stat main5static.o -L. -lmy_array -Wl,-rpath,.

	$(CC) $(FLAGS) -o run-dyn main5.o -ldl

main5static.o: main5static.c

	$(CC) -c $(FLAGS) main5static.c


main5.o: main5.c

	$(CC) -c $(FLAGS) main5.c
```

```
arr.o: arr.c
	$(CC) -c -fPIC $(FLAGS) arr.c
libmy_array.so: arr.o
	$(CC) $(FLAGS) -shared -o libmy_array.so arr.o
clean:
	rm -f *.o run-stat run-dyn *.so
```

arr.h

```c
#ifndef _arr_H_
#define _arr_H_

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <stdint.h>

typedef int32_t ElemType;

typedef struct Massive{
    int size;
    ElemType* data;
} Massive;

void create_massive(Massive* mass, int Lenght);
void delete_massive(Massive* mass);
int length_massive(Massive* mass);
ElemType* get_elem_massive(Massive* mass, int index);
bool empty_massive(Massive* mass);
void print_massive(Massive* mass);

#endif
```

arr.c

```c

#include "/Users/amelia/Documents/arr.h"
#include <errno.h>


void create_massive(Massive* mass, int Length){
    if(Length >= 0){
        mass->size = Length;
        mass->data = malloc(sizeof(ElemType) * Length);
```

```
        printf("Massive created!\n");
    } else{
        printf("Error! Wrong size of massive!\n");
        exit(EXIT_FAILURE);
    }
    return;
}

void delete_massive(Massive* mass){
    mass->size = 0;
    free(mass->data);
    mass->data = NULL;
}

ElemType* get_elem_massive(Massive* mass, int index){
    if (index > mass->size || index < 0) {
        printf("Error: wrong index \n");
        return NULL;
    }
    ElemType* curr = mass->data;
    int k = 0;
    while (k < index) {
        k++;
        curr = curr + sizeof(ElemType);
    }
    return (curr);
}

int lenght_massive(Massive* mass){
    return mass->size;
}

bool empty_massive(Massive* mass){
    return (mass->size == 0);
}

void print_massive(Massive* mass){
    printf("Massive:\n Elems:%d \n|", mass->size);
    ElemType* curr = mass->data;
    for (int i = 0; i < mass->size; i++) {
        printf("%d|", *curr);
        curr = curr + sizeof(ElemType);
    }
    printf("|\n");
    return;
}
```

main5.c

```
#include "/Users/amelia/Documents/arr.h"

#include <dlfcn.h>
#include <errno.h>
#include <stdio.h>
```

```c
#include <stdlib.h>

#define EXPEREMENTAL_SIZE 10

int main(int argc, char const *argv[]) {
    Massive mass;
    void* handle = dlopen ("libmy_array.so", RTLD_LAZY);
    if (!handle) {
        fputs (dlerror(), stderr);
        exit(1);
    }
    void (*creater)(Massive*, int);
    creater = dlsym(handle, "create_massive");
    (*creater)(&mass, EXPEREMENTAL_SIZE);
    ElemType* (*getter)(Massive*, int);
    getter = dlsym(handle, "get_elem_massive");
    int (*lenghter)(Massive*);
    lenghter = dlsym(handle, "lenght_massive");
    for(int i=0; i < (*lenghter)(&mass); i++){
        *((*getter)(&mass, i)) = i;
    }
    *((*getter)(&mass, 0)) = 233;
    void (*voider)(Massive*);
    voider = dlsym(handle, "print_massive");
    (*voider)(&mass);
    void (*resizer)(Massive*, int);
    (*voider)(&mass);
    voider = dlsym(handle, "delete_massive");
    (*voider)(&mass);
    return 0;
}
```

main5static.c

```c
#include "/Users/amelia/Documents/arr.h"

#define EXPEREMENTAL_SIZE 10

//Бронников Максим
//Использование библиотеки, запущенной на этапе линковки

int main(int argc, char const *argv[]) {
    Massive mass;
    create_massive(&mass, EXPEREMENTAL_SIZE);
    for(int i=0; i < lenght_massive(&mass); i++){
        *(get_elem_massive(&mass, i)) = i;
    }
    *(get_elem_massive(&mass, 0)) = 233;
    print_massive(&mass);
    *(get_elem_massive(&mass, EXPEREMENTAL_SIZE - 1)) = 98;
    print_massive(&mass);
```

```
    delete_massive(&mass);
    return 0;
}
```

4. Вывод

В простых программах с минимальной функциональностью статические библиотеки могут быть предпочтительнее. В программах же, использующих несколько библиотек, применение совместно используемых библиотек позволяет снизить потребление оперативной и дисковой памяти во время работы приложения. Это достигается за счет того, что одна совместно используемая библиотека может использоваться одновременно несколькими приложениями, при этом она присутствует в памяти в единственном экземпляре. В случае со статическими библиотеками каждая программа загружает свою собственную копию библиотечных функций.

В GNU/Linux доступно два метода работы с совместно используемыми библиотеками (оба метода берут свое начало в Sun Solaris). Первый способ – это динамическая компоновка вашего приложения с совместно используемой библиотекой. При этом загрузку библиотеки при запуске программы возьмет на себя Linux (если, конечно, она не была загружена в память раньше). Второй способ подразумевает явный вызов функций библиотеки в процессе т. н. **динамической загрузки**. В этом случае программа явно загружает нужную библиотеку, а затем вызывает определенную библиотечную функцию. На этом методе обычно основан механизм загрузки подключаемых программных модулей – плагинов. В этом случае приложение само "решает", какие библиотеки загрузить, после чего вызывает библиотечные функции, как если бы они были частью исходной программы.