Задача 1

Статические библиотеки

Библиотека компилируется в объектный файл, весь объектный код библиотеки включается в конечный исполняемый файл

- увеличивается размер исполняемого файла
- при изменении библиотеки придется компилировать проект заново
- исполняемый файл будет содержать в себе все необходимое и его можно будет распространять на другие устройства
- код библиотеки содержится в исполняемом файле в секции .text вместе с кодом основных файлов

```
//hello1.c
#include "hello_static.h"

int main() {
	hello_from_static_lib();
	return 0;
}
```

```
//hello_static.c
#include <stdio.h>

void hello_from_static_lib() {
	printf("Hello static world!\n");
}
```

```
//hello_static.h
void hello_from_static_lib();
```

Компилируем библиотеку без сборки и получаем объектный файл

gcc -c hello_static.c -o hello_static.o

Запаковываем объектный файл библиотеки в статическую библиотеку (расширение *.a)

ar - заархивировать

r - переупаковать архив вместе с новыми файлами

c - создать архив, если его не существует

ar rc libhello_static.a hello_static.o (+все остальные объектные файлы библиотеки)

ar -t libhello_static.a - посмотреть, что входит в библиотеку

Первый способ компиляции и запуска:

gcc hello1.c libhello_static.a -o hello1

./hello1

Второй способ компиляции и запуска:

gcc hello1.c -L. -lhello_static -o hello1
./hello1

Название библиотеки начинается с lib!!!!!!!!!!!!!!!
-L. - библиотека находится в текущей директории
если библиотека в другой директории, то `-L path/to/libdir`

Динамические библиотеки

- разные программы могут использовать одну копию библиотеки
- можно обновлять и не придется перекомпилировать все исполняемые файлы
- для запуска нужно установить библиотеку

```
//hello2.c
#include <stdio.h>
#include "hello_dynamic.h"

int main() {
	hello_from_dynamic_lib();
	return 0;
}
```

```
//hello_dynamic.c
#include <stdio.h>

void hello_from_dynamic_lib() {
	printf("Hello dynamic world!\n");
}
```

```
//hello_dynamic.h
void hello_from_dynamic_lib();
```

Компилируем библиотеку без линковки и получаем объектный файл:

gcc -c hello_dynamic.c -o hello_dynamic.o

Создаем shared object с помощью флага -shared (происходит линковка динамической библиотеки):

gcc -shared -o libhello_dynamic.so hello_dynamic.o

Перемещаем созданную библиотеку в директорию со всеми либами:

sudo cp libhello_dynamic.so /usr/lib

Вместо этого можно написать:

export LD_LIBRARY_PATH=./

(установить новую переменную окружения с текущей директорией)

Компилируем и запускаем:

gcc hello2.c -fPIC -L. -lhello_dynamic -o hello
./hello2

// -fPIC флаг для генерации исходного кода в позиционно-независимый (используется только относительная адресация, не абсолютная), который не будет зависеть от расположения в виртуальном адресном пространстве
благодаря этому есть возможность не загружать библиотеку на стадии линковки, а подгружать части непосредственно во время выполнения

Загружаемые библиотеки

Создаем динамическую библиотеку точно так же, как выше
Переносим ее в директорию с либами:
sudo cp libhello_dynamic.so /usr/lib
или
export LD_LIBRARY_PATH=./

Добавляем этот заголовочный файл в главный файл hello3.c:
#include <dlfcn.h>

```
//hello3.c
#include <stdio.h>
#include <dlfcn.h>
#include <stdlib.h>

int main() {
	void* handle; //описатель
	void (*hello_from_dyn_runtime_lib)(void); //создаем пустой указатель на функцию с нужными аргументами и возвращаемым значением
	handle = dlopen("libhello_runtime.so", RTLD_LAZY); //загружаем динамическую библиотеку с флагом позднего связывания (ищутся только те символы, которые используются в программе)
	if (!handle) {
		fprintf(stderr, "%s\n", dlerror());
		exit(EXIT_FAILURE); //если не открылось, выдаем ошибку
	}
	dlerror(); //очищаем ошибки

	hello_from_dyn_runtime_lib = (void (*)(void)) dlsym(handle, "hello_from_dyn_runtime_lib"); //присваиваем в наш заранее созданный указатель нужную функцию из библиотеки
	if (dlerror() != NULL) {
		fprintf(stderr, "%s\n", error);
		exit(EXIT_FAILURE); //если функция не нашлась, то падаем  с ошибкой
	}

	hello_from_dyn_runtime_lib(); //вызываем найденную функцию
	dlclose(handle); //закрываем описатель
```

```
	return 0;
}
```

```
//hello_runtime.c
#include <stdio.h>

void hello_from_dyn_runtime_lib() {
	printf("Hello dynamic runtime world!\n");
}
```

Компилируем и запускаем:

gcc hello3.c -ldl -o hello3

./hello3

// **-ldl** это обозначение библиотеки для компоновщика. Он говорит компоновщику найти и связать файл с именем libdl.so (или иногда libdl.a). Это имеет тот же эффект, что и размещение полного пути к рассматриваемой библиотеке в той же позиции командной строки.

- **Функция dlopen() загружает файл динамического общего объекта (разделяемой библиотеки), код которого должен быть ПОЗИЦИОННО-НЕЗАВИСИМЫМ**

Анализ ELF-файлов

objdump -t *.so //выводит таблицу символов
readelf -h *.exe //заголовок исполняемого файла(entry point - адрес точки входа)
readelf -h *.so //заголовок исполняемого файла(entry point - мусор, поэтому не запустится)
readelf -S *.elf //секции файла
file *.elf //информация о файле(статический или динамический)
ldd *.elf //посмотреть внешние зависимости
nm *.elf //посмотреть все имена и их области видимости

ELF-file:

- elf-заголовок
- программные заголовки
  нужны для формирования процесса

Файл ELF состоит из нуля или более сегментов, и описывает, как создать процесс, образ памяти для исполнения в рантайме. Когда ядро видит эти сегменты, оно размещает их в виртуальном адресном пространстве, используя системный вызов mmap(2). Другими словами, конвертирует заранее подготовленные инструкции в образ в памяти.

- секции

Заголовки секции определяют все секции файла. Как уже было сказано, эта информация используется для линковки и релокации.
Секции появляются в ELF-файле после того, как компилятор GNU C преобразует код C в ассемблер, и ассемблер GNU создаёт объекты.

Как показано на рисунке вверху, сегмент может иметь 0 или более секций. Для исполняемых файлов существует четыре главных секций: .text, .data, .rodata, и .bss. Каждая из этих секций загружается с различными правами доступа.

ldd hello:

1) **linux.vdso.so.1**

Это виртуальный общий объект, у которого нет физического файла на диске; это часть ядра, которая экспортируется в адресное пространство каждой программы при загрузке.

Основная цель - повысить эффективность вызова определенных системных вызовов

2) **libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f29ea4c1000)** - стандартная библиотека Си
3) **/lib64/ld-linux-x86-64.so.2 (0x00007f29ea6c3000)** - динамический линковщик

Переменные окружения

bash(init ENV)
prog1(init ENV) v -> change

bash(init ENV)

чтобы при каждом запуске bash переменная была другой, нужно добавить в **.bashrc**
export PATH="$PATH:/usr/sbin"
не писать ./

setenv(... -> …) только в prog1, потом сбрасывается
printf()...

ulimit -a все системные лимиты (максимальное число процессов и тд)

Права

right file

при открытии файла происходит системный вызов open(...)
можно блокировать рекомендательно или насовсем
дальше смотрит в файловой системе, какие у файла права

lsattr посмотреть атрибуты файлов
chattr изменить атрибуты файла

ps ax | grep hello посмотреть запущенные процессы и PIDs
strace -p 146 все системные вызовы данного процесса

Системные вызовы

malloc
sbrk
printf
write
открытие файла
open
порождение процесса
fork
clone
общение между процессами
socket
pipe
signal
sem (семафоры)

Дочерние процессы **fork**

наследует:
ENV
PGID (process group ID) //**группа процессов??? man ps**
из под кого запущено
cow (копируются массивы данных только тогда, когда начинаются изменения)
не наследует:
PID
таблицу блокировок bloc

Отложенная запись

prog1 (fd) edit file
prog2(fd1) rm file
удаляется после закрытия prog1 (fd отпускается)

Ссылки файловой системы

символьные и жесткие
inode - циферки для каждого файла
ls -i
у жестких ссылок одинаковый inode

Изменить ссылку:
ln source dest для символьных
ln -s source dest для жестких

у директорий много жестких ссылок . и ..
сама на себя через .
поддиректории через ..

ls -d lab1/* | wc -l

Генерация файла

for i in {1..1000}; do `echo “line $i” >> lines.txt`; done

awk
sed

Как найти код?

uname -a
/etc os-release там можно найти версию операционной системы

Задача 2

1. a) i)

```
#include <stdio.h>

int main() {
	printf("Hello world!\n");
	return 0;
}
```

gcc hello.c -o hello
strace ./hello

execve("./hello", ["./hello"], 0x7fff100497d0 /* 25 vars */) = 0	**int execve(const char *filename, char *const argv[], char *const envp[]);** *execve*() выполняет программу, задаваемую аргументом filename *argv* — это массив строковых параметров, передаваемых новой программе *envp* — это массив строк в формате ключ=значение, которые передаются новой программе в качестве окружения (environment). Оба массива *argv* и *envp* завершаются указателем null
brk(NULL) = 0x55c52d9ff000	**int brk(void *addr);** изменяет расположение маркера окончания программы (program break), который определяет конец сегмента данных процесса (т.е., маркер окончания --- это первая точка после конца сегмента неинициализированных данных). Увеличение маркера окончания

	программы позволяет процессу выделить память; уменьшение маркера приводит к освобождению памяти. Системный вызов Linux в случае успешного завершения возвращает новый маркер окончания программы
arch_prctl(0x3001 /* ARCH_??? */, 0x7fffdffb5680) = -1 EINVAL (Invalid argument)	**int arch_prctl(int code, unsigned long *addr);** задаёт состояние процесса или нити, зависящие от архитектуры
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)	**int access(const char *pathname, int mode);** проверяет, имеет ли вызвавший процесс права доступа к файлу pathname
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3	**int openat(int dirfd, const char *pathname, int flags);** Получив в pathname имя файла, open() возвращает файловый дескриптор --- небольшое, неотрицательное значение --- для использования в последующих системных вызовах (read(2), write(2), lseek(2), fcntl(2) и т.д.) Если в pathname задан относительный путь, то он считается относительно каталога, на который ссылается файловый дескриптор dirfd (а не относительно текущего рабочего каталога вызывающего процесса, как это делается в open()). Если в pathname задан относительный путь и dirfd равно специальному значению AT_FDCWD, то pathname рассматривается относительно текущего рабочего каталога вызывающего процесса (как open()). Если в pathname задан абсолютный путь, то dirfd игнорируется.
fstat(3, {st_mode=S_IFREG|0644, st_size=33967, ...}) = 0	**int stat(const char *pathname, struct stat *buf);** возвращает информацию о файле в буфер, на который указывает buf
mmap(NULL, 33967, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fecccf92000	**void *mmap2(void *addr, size_t length, int prot,** **int flags, int fd, off_t pgoffset);** fd - файловый дескриптор выделяет память в виртуальном адресном пространстве, если передан null, то произвольно, иначе опирается на переданный адрес

	prot - уровень защиты памяти возвращает указатель на выделенную память или код ошибки
close(3) = 0	**int close(int fd);** закрывает файловый дескриптор, который после этого не ссылается ни на один и файл и может быть использован повторно
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3	
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\300A\2\0\0\0\0"..., 832) = 832	**ssize_t read(int fd, void *buf, size_t count);** Вызов read() пытается прочитать count байт из файлового дескриптора fd в буфер, начинающийся по адресу buf. позиция в файле увеличивается на значение count
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784 pread64(3, "\4\0\0\0\20\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0", 32, 848) = 32 pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\30x\346\264ur\f|Q\226\236i\253-'o"..., 68, 880) = 68	**ssize_t pread(int fd, void *buf, size_t count, off_t offset);** pread() читает максимум count байтов из файлового дескриптора fd, начиная со смещения offset (от начала файла), в буфер, начиная с buf. Текущая позиция файла не изменяется.
fstat(3, {st_mode=S_IFREG|0755, st_size=2029592, ...}) = 0	
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fecccf90000	
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784 pread64(3, "\4\0\0\0\20\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0", 32, 848) = 32 pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\30x\346\264ur\f|Q\226\236i\253-'o"..., 68, 880) = 68	

mmap(NULL, 2037344, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7fecccd9e000	
mmap(0x7fecccdc0000, 1540096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x22000) = 0x7fecccdc0000 mmap(0x7fecccf38000, 319488, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x19a000) = 0x7fecccf38000 mmap(0x7fecccf86000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1e7000) = 0x7fecccf86000 mmap(0x7fecccf8c000, 13920, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7fecccf8c000	
close(3)= 0	
arch_prctl(ARCH_SET_FS, 0x7fecccf91540) = 0	
mprotect(0x7fecccf86000, 16384, PROT_READ) = 0 mprotect(0x55c52c720000, 4096, PROT_READ) = 0 mprotect(0x7fecccfc8000, 4096, PROT_READ) = 0	должна изменить защиту доступа на ту, которая указана prot для целых страниц, содержащих любую часть адресного пространства процесса, начиная с адреса addr и продолжая для байтов len
munmap(0x7fecccf92000, 33967) = 0	**int munmap(void *addr, size_t length);** освобождение памяти по адресу
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}) = 0	
brk(NULL) = 0x55c52d9ff000 brk(0x55c52da20000) = 0x55c52da20000	
write(1, "Hello world!\n", 13Hello world!) = 13	**ssize_t write(int fd, const void *buf, size_t count);** вывод текста
exit_group(0) = ? +++ exited with 0 +++	выход из всех потоков в процессе Этот системный вызов эквивалентен _exit(2), за исключением того, что он завершает не только вызывающий поток,

	но и все потоки в группе потоков вызывающего процесса. нет возвращаемого значения

1. a) ii) используйте write в программе вместо printf()

```
#include <unistd.h>

int main() {
	write(1, "Hello world!\n", 13);
	return 0;
}
```

1. a) iii) напишите свою обертку над этим сисколом

```
#include <unistd.h>
#include <sys/syscall.h>

void mywrite(int fd, const void *buf, size_t count) {
	syscall(SYS_write, fd, buf, count);
}


int main() {
	mywrite(1, "Hello syscall world!\n", 21);
	return 0;
}
```

long syscall(long number, ...);

syscall() - это небольшая библиотечная функция, которая вызывает системный вызов, интерфейс которого на языке ассемблера имеет указанный номер с указанными аргументами
Символьные константы для номеров системных вызовов можно найти в заголовочном файле <sys/syscall.h> (**/usr/include/x86_64-linux-gnu/asm/unistd_64.h**)

Возвращаемое значение определяется вызываемым системным вызовом. Как правило, возвращаемое значение 0 указывает на успех. Возвращаемое значение -1 указывает на ошибку, а номер ошибки сохраняется в errno.

2. Напишите код, который напечатает hello world без использования функции syscall().

С вызовом syscall:

```
.data
    hello:
        .ascii "Hello world!\n"
        len = . - hello

.text
    .global _start
    _start:
        mov     $1, %rax
        mov     $1, %rdi
        mov     $hello, %rsi
        mov     $13, %rdx
        syscall

        mov     $60, %rax
        xor     %rdi, %rdi
        syscall
```

gcc -c hello_asm.s
ld -o asm hello_asm.o

strace ./asm

```
execve("./asm", ["./asm"], 0x7ffeb9cf5840 /* 25 vars */) = 0
write(1, "Hello world!\n", 13Hello world!
)         = 13
exit(0)                                 = ?
+++ exited with 0 +++
```

С прерыванием:

```
.data
    hello:
        .ascii "Hello world!\n"
        len = . - hello

.text
    .global _start
    _start:
        mov     $4, %rax
        mov     $1, %rbx
        mov     $hello, %rcx
        mov     $len, %rdx
        int $0x80

        mov     $1, %rax
        xor     %rdi, %rdi
        int $0x80
```

strace ./int

```
execve("./int", ["./int"], 0x7ffd1e2db990 /* 25 vars */) = 0
strace: [ Process PID=412 runs in 32 bit mode. ]
write(1, "Hello world!\n", 13Hello world!
)         = 13
exit(1)                                 = ?
+++ exited with 1 +++
```

b) Запустите под strace команду ‘wget kernel.org’ (если нет wget, используйте curl). Получите статистику использования системных вызовов порожденным процессом.

strace -c wget kernel.org

```
URL transformed to HTTPS due to an HSTS policy
--2023-02-21 23:31:38--  https://kernel.org/
Resolving kernel.org (kernel.org)... 139.178.84.217, 2604:1380:4641:c500::1
Connecting to kernel.org (kernel.org)|139.178.84.217|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 15564 (15K) [text/html]
Saving to: ‘index.html.3’

index.html.3
100%[=============================================>]  15.20K  --.-KB/s
in 0.002s

2023-02-21 23:31:40 (6.23 MB/s) - ‘index.html.3’ saved [15564/15564]

% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 12.94    0.004119          67        61           read
  9.79    0.003118          51        61        15 openat
  9.04    0.002877          35        82           mmap
  8.95    0.002848          91        31           write
  6.88    0.002190          42        52           close
  6.22    0.001980          38        51           fstat
  6.10    0.001942          62        31           ioctl
  4.80    0.001527          61        25           getpgrp
  4.53    0.001441          60        24           getpid
  3.61    0.001148          67        17         7 stat
  3.60    0.001145          76        15           rt_sigaction
  3.20    0.001020          44        23           futex
  2.96    0.000942         134         7           socket
  2.82    0.000897          44        20           mprotect
  2.40    0.000765          95         8           setitimer
  2.25    0.000716         119         6         3 connect
```

1.12	0.000356	178	2		select
1.10	0.000351	70	5		rt_sigprocmask
0.82	0.000260	52	5		brk
0.71	0.000225	45	5		getuid
0.69	0.000221	221	1		sendmmsg
0.69	0.000220	73	3		recvmsg
0.60	0.000191	63	3		poll
0.57	0.000183	61	3		munmap
0.50	0.000158	39	4		lseek
0.36	0.000116	116	1		ftruncate
0.27	0.000087	43	2		recvfrom
0.24	0.000078	9	8		pread64
0.24	0.000076	38	2		getsockname
0.18	0.000057	57	1		statfs
0.18	0.000057	57	1		getrandom
0.16	0.000052	52	1		flock
0.15	0.000049	49	1		uname
0.15	0.000048	48	1		sendto
0.15	0.000048	48	1		setsockopt
0.15	0.000047	47	1		utime
0.14	0.000045	22	2	1	arch_prctl
0.13	0.000041	41	1		getgroups
0.13	0.000040	40	1		bind
0.13	0.000040	40	1		sysinfo
0.13	0.000040	40	1		set_tid_address
0.12	0.000038	38	1		set_robust_list
0.12	0.000038	38	1		prlimit64
0.00	0.000000	0	1	1	access
0.00	0.000000	0	1		execve
------	----------	----------	---------	---------	----------
100.00	0.031837		575	27	total

3. ptrace для вывода системных вызовов дочернего процесса

```
long  int ptrace(enum __ptrace_request request, pid_t pid,
                 void * addr, void * data);
```

request:

PTRACE_TRACEME - начать трассировку дочернего процесса

PTRACE_ATTACH - начать трассировку существующего процесса

PTRACE_DETACH - завершение трассировки (восстанавливает исходное состояние процесса, возвращается ссылка на настоящего родителя)

PTRACE_PEEKTEXT, PTRACE_PEEKDATA - прочитать слово по адресу addr из адресного пространства дочернего процесса

PTRACE_GETREGS — читает текущее состояние регистров процесса в структуру user_regs_struct

PTRACE_PEEKUSER - читается слово из структуры user

PTRACE_SYSCALL - возобновить работу дочернего процесса до следующего системного вызова

PTRACE_SINGLESTEP - возобновить работу дочернего процесса до следующей инструкции

PTRACE_CONT - просто возобновляет работу трассируемого процесса

PTRACE_KILL - проверяет -- "жив" ли трассируемый процесс, затем устанавливает код завершения дочернего процесса в значение sigkill, сбрасывает бит пошаговой трассировки и активирует дочерний процесс, который в соответствии с кодом завершения прекращает свою работу.

Чтобы как-то различать системные вызовы и другие остановки (например SIGTRAP), предусмотрен специальный параметр PTRACE_O_TRACESYSGOOD — при остановке на системном вызове родительский процесс получит в статусе SIGTRAP | 0x80:

ptrace(PTRACE_SETOPTIONS, pid, 0, PTRACE_O_TRACESYSGOOD);

Функция **wait** приостанавливает выполнение текущего процесса до тех пор, пока дочерний процесс не завершится, или до появления сигнала, который либо завершает текущий процесс, либо требует вызвать функцию-обработчик. Если дочерний процесс к моменту вызова функции уже завершился (так называемый "зомби" ("zombie")), то функция немедленно возвращается. Системные ресурсы, связанные с дочерним процессом, освобождаются.

Функция **waitpid** приостанавливает выполнение текущего процесса до тех пор, пока дочерний процесс, указанный в параметре pid, не завершит выполнение, или пока не появится сигнал, который либо завершает текущий процесс либо требует вызвать функцию-обработчик. Если указанный дочерний процесс к моменту вызова функции уже завершился (так называемый "зомби"), то функция немедленно возвращается. Системные ресурсы, связанные с дочерним процессом, освобождаются.

Параметр pid может принимать несколько значений:

< -1

означает, что нужно ждать любого дочернего процесса, идентификатор группы процессов которого равен абсолютному значению pid.

-1

означает ожидание любого дочернего процесса; функция wait ведет себя точно так же.

0

означает ожидание любого дочернего процесса, идентификатор группы процессов которого равен идентификатору текущего процесса.

> 0
означает ожидание дочернего процесса, чей идентификатор равен pid.

WIFEXITED(status)
не равно нулю, если дочерний процесс успешно завершился.
WEXITSTATUS(status)
возвращает восемь младших битов значения, которое вернул завершившийся дочерний процесс. Эти биты могли быть установлены в аргументе функции exit() или в аргументе оператора return функции main(). Этот макрос можно использовать, только если WIFEXITED вернул ненулевое значение.
WIFSIGNALED(status)
возвращает истинное значение, если дочерний процесс завершился из-за необработанного сигнала.
WTERMSIG(status)
возвращает номер сигнала, который привел к завершению дочернего процесса. Этот макрос можно использовать, только если WIFSIGNALED вернул ненулевое значение.
WIFSTOPPED(status)
возвращает истинное значение, если дочерний процесс, из-за которого функция вернула управление, в настоящий момент остановлен; это возможно, только если использовался флаг WUNTRACED или когда подпроцесс отслеживается (см. ptrace(2)).
WSTOPSIG(status)
возвращает номер сигнала, из-за которого дочерний процесс был остановлен. Этот макрос можно использовать, только если WIFSTOPPED вернул ненулевое значение.

регистр rax в момент остановки заменен, и поэтому необходимо использовать сохраненный state.orig_rax:

```
#include <sys/ptrace.h>
#include <unistd.h>
#include <sys/wait.h>
#include <errno.h>
#include <stdio.h>
#include </usr/include/x86_64-linux-gnu/sys/user.h>

void child() {
	ptrace(PTRACE_TRACEME, 0, 0, 0);
	execl("/bin/echo", "/bin/echo", "Hello, world!", NULL);
}

void parent(pid_t pid) {
	int status;
	waitpid(pid, &status, 0);
	ptrace(PTRACE_SETOPTIONS, pid, 0, PTRACE_O_TRACESYSGOOD);

	while (!WIFEXITED(status)) {

		struct user_regs_struct state;

		ptrace(PTRACE_SYSCALL, pid, 0, 0);
```

```
            waitpid(pid, &status, 0);

            // at syscall
            if (WIFSTOPPED(status) && (WSTOPSIG(status) & 0x80)) {
                ptrace(PTRACE_GETREGS, pid, 0, &state);
                printf("SYSCALL %lld at %08llx\n", state.orig_rax, state.rip);

                // skip after syscall
                ptrace(PTRACE_SYSCALL, pid, 0, 0);
                waitpid(pid, &status, 0);
            }
        }
}

int main(int argc, char *argv[]) {
        pid_t pid = fork();
        if (pid)
                parent(pid);
        else
                child();
        return 0;
}
```

Задача 3

part 1

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <unistd.h>
#include <dirent.h>
#include <string.h>
#include <sys/stat.h>


size_t find_name_begin(char* name, size_t name_size) {
    size_t pos = 0;
    while (pos < name_size && name[pos] != '/') {
        ++pos;
    }
    if (name[pos] == '/') ++pos;
    if (pos == name_size) pos = 0;
    return pos;
}

size_t find_name_end(char* name, int name_begin, size_t name_size) {
    size_t pos = name_begin;
```

```
    while (pos < name_size && name[pos] != '.') {
        ++pos;
    }

    return pos;
}

void reverse(char* begin, char* end) {
    for (; begin < end; ++begin, --end) {
        char tmp = *begin;
        *begin = *end;
        *end = tmp;
    }
}

void get_reverse_file_name(char* file_name, char** reverse_name) {
    size_t name_size = strlen(file_name);
    (*reverse_name) = (char*)malloc(name_size+1);
    strcpy((*reverse_name), file_name);

    size_t name_begin = 0;
    size_t name_end = find_name_end(file_name, name_begin, strlen(file_name));
    reverse((*reverse_name), (*reverse_name) + name_end - 1);
    (*reverse_name)[name_size] = '\0';
}

void write_reverse_file(FILE* fin, char* reverse_name) {
    FILE* fout = fopen(reverse_name, "w");

    fseek(fin, 0, SEEK_END);
    int size = ftell(fin);

    while ((--size) >= 0) {
        fseek(fin, size, SEEK_SET);
        char cur = fgetc(fin);
        fputc(cur, fout);
    }

    fclose(fout);
}

void create_dir(char *path_name) {
    size_t path_name_size = strlen(path_name);
    size_t name_begin = find_name_begin(path_name, path_name_size);
    size_t name_end = find_name_end(path_name, name_begin, path_name_size);
    size_t format = path_name_size - name_end;

    DIR *source_dir_pointer;
    struct dirent *current;
    struct stat stat_buf;

    if (!stat(path_name, &stat_buf) && S_ISDIR(stat_buf.st_mode)) {
    } else {
        fprintf(stderr, "Specified directory doesn't exist\n");
```

```
    return;
}

if ((source_dir_pointer = opendir(path_name)) == NULL) {
    fprintf(stderr, "Can`t open directory %s\n", path_name);
    return;
}

if (chdir(path_name) == -1) printf("Directory %s unavailable\n", path_name);
chdir("..");

size_t dir_name_size = path_name_size - name_begin+1;
char* dir_name = (char*)malloc(dir_name_size);
char* reverse_dir_name = (char*)malloc(dir_name_size);
for (size_t i = 0; i < dir_name_size; ++i) {
    reverse_dir_name[i] = path_name[name_begin+i];
    dir_name[i] = path_name[name_begin+i];
}
reverse_dir_name[dir_name_size] = '\0';

reverse(reverse_dir_name, reverse_dir_name+dir_name_size-2-format);
reverse_dir_name[dir_name_size] = '\0';


if (mkdir(reverse_dir_name, 0777) != 0) {
    printf("Directory %s already exists\n", reverse_dir_name);
} else {
    printf("Directory %s was created!\n", reverse_dir_name);
}

if (chdir(dir_name) == -1) fprintf(stderr, "Directory %s unavailable\n", dir_name);
current = readdir(source_dir_pointer);
while (current != NULL) {
    stat(current->d_name, &stat_buf);
    if (S_ISREG(stat_buf.st_mode)) {
        char* reverse_name;
        get_reverse_file_name(current->d_name, &reverse_name);

        FILE* fin = fopen(current->d_name, "r");
        if (fin == NULL) {
            perror(current->d_name);
            exit(-1);
        }

        chdir("..");
        chdir(reverse_dir_name);

        write_reverse_file(fin, reverse_name);
        fclose(fin);
        if (reverse_name != NULL) free(reverse_name);
        chdir("..");
        if (chdir(dir_name) == -1) printf("Directory %s unavailable\n", dir_name);
    }
    current = readdir(source_dir_pointer);
```

```
    }
    chdir("..");
    closedir(source_dir_pointer);
}

int main(int argc, char** argv) {

    if (argc < 2) {
        fprintf(stderr, "Directory name not specified\n");
        return 0;
    }

    create_dir(argv[1]);
    return 0;
}
```

part 2

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>
#include <sys/stat.h>


void create_dir(char* dir_name) {
    struct stat stat_buf;

    if (mkdir(dir_name, 0777) != 0) {
        printf("Directory already exists\n");
    } else {
        printf("Directory was created!\n");
    }
    return;
}

void print_dir(char* dir_name) {
    DIR *dir_pointer;
    struct dirent *current;

    if ((dir_pointer = opendir(dir_name)) == NULL) {
        fprintf(stderr, "Can`t open directory %s\n", dir_name);
        return;
    }

    chdir(dir_name);
    current = readdir(dir_pointer);
    while (current != NULL) {
```

```
        printf("%s\n", current->d_name);
        current = readdir(dir_pointer);
    }
    chdir("..");
    closedir(dir_pointer);
}

int dir_is_empty(DIR* dir_pointer) {
    int files_found = 0;
    struct dirent * dirent;
    while(dirent = readdir(dir_pointer)) {
        if(strcmp(dirent->d_name, ".") || strcmp(dirent->d_name, "..")) {
            files_found = 1;
            break;
        }
    }
    return !files_found;
}

void delete_dir(char* dir_name) {
    DIR* dir_pointer;
    struct dirent *current;
    struct stat stat_buf;

    if ((dir_pointer = opendir(dir_name)) == NULL) {
        fprintf(stderr, "Can`t open directory %s\n", dir_name);
        return;
    }

    chdir(dir_name);
    current = readdir(dir_pointer);
    while (!dir_is_empty(dir_pointer) && current != NULL) {
        stat(current->d_name, &stat_buf);
        if (S_ISDIR(stat_buf.st_mode)) {
            if (strcmp(current->d_name, ".") == 0 || strcmp(current->d_name, "..") == 0 )
                continue;
            delete_dir(current->d_name);
        } else {
            if (remove(current->d_name) != 0) {
                printf("Can't delete file %s\n", current->d_name);
            } else {
                printf("File %s was deleted\n", current->d_name);
            }
        }
        current = readdir(dir_pointer);
    }

    chdir("..");
    closedir(dir_pointer);
    if (rmdir(dir_name) == -1) {
        printf("Can't delete directory %s\n", dir_name);
    } else {
        printf("Directory %s was deleted\n", dir_name);
    }
```

```
}

void create_file(char* file_name) {
    FILE* f = fopen(file_name, "w");
    fclose(f);
    printf("File %s created\n", file_name);
}

void print_file(char* file_name) {
    FILE* f = fopen(file_name, "r");
    if (f == NULL) {
        fprintf(stderr, "Impossible to read file %s\n", file_name);
    }

    fseek(f, 0, SEEK_END);
    size_t size = ftell(f);
    fseek(f, 0, SEEK_SET);

    for (size_t i = 0; i < size; ++i) {
        char cur;
        cur = getc(f);
        printf("%c", cur);
    }
    printf("\n");

    fclose(f);
}

void delete_file(char* file_name) {
    if (remove(file_name) != 0) {
        fprintf(stderr, "Impossible to delete file %s\n", file_name);
    } else {
        printf("File %s was deleted\n", file_name);
    }
}

void create_sym_link(char* file_name) {
    if (symlink(file_name, "symlink") != 0) {
        fprintf(stderr, "Impossible to create sym link %s\n", file_name);
    } else {
        printf("Sym link to file %s was created\n", file_name);
    }
}

void print_sym_link(char* link_name) {
    char buf[100];
    size_t size = readlink(link_name, buf, 100);
    if (size == -1) {
        fprintf(stderr, "Impossible to read sym link %s\n", link_name);
        return;
    }
    write(1, buf, size);
    printf("\n");
}
```

```
void print_file_sym_link(char* link_name) {
    FILE* f = fopen(link_name, "r");
    if (f == NULL) {
        fprintf(stderr, "Impossible to read file %s\n", link_name);
    }

    fseek(f, 0, SEEK_END);
    size_t size = ftell(f);
    fseek(f, 0, SEEK_SET);

    for (size_t i = 0; i < size; ++i) {
        char cur;
        cur = getc(f);
        printf("%c", cur);
    }
    printf("\n");

    fclose(f);
}

void delete_sym_link(char* link_name) {
    if (unlink(link_name) != 0) {
        fprintf(stderr, "Impossible to delete sym link %s\n", link_name);
    } else {
        printf("Sym link %s was deleted\n", link_name);
    }
}

void create_hard_link(char* file_name) {
    if (link(file_name, "hardlink") != 0) {
        fprintf(stderr, "Impossible to create hard link %s\n", file_name);
    } else {
        printf("Hrd link to %s was created\n", file_name);
    }
}

void delete_hard_link(char* link_name) {
    if (unlink(link_name) != 0) {
        fprintf(stderr, "Impossible to delete hard link %s\n", link_name);
    } else {
        printf("Hard link %s was deleted\n", link_name);
    }
}

void get_mode(char* file_name) {
    struct stat stat_buf;
    stat(file_name, &stat_buf);
    mode_t mode = stat_buf.st_mode;

    char str_mode[10];
    const char all_mode[] = "rwxrwxrwx";
    for (size_t i = 0; i < 9; ++i) {
        str_mode[i] = (mode & (1 << (8-i))) ? all_mode[i] : '-';
```

```
    }
    str_mode[9] = '\0';

    printf("mode: %s\n", str_mode);
    printf("number of hard links: %ld\n", stat_buf.st_nlink);
}

void change_file_mode(char* file_name, char* str_mode) {
    mode_t mode;
    mode = strtol(str_mode, 0, 8);

    if (chmod(file_name, mode) < 0) {
        fprintf(stderr, "Impossible to set mode %s for file %s\n", str_mode, file_name);
        return;
    } else {
        printf("Mode for file %s was changed\n", file_name);
    }
}

int main(int argc, char** argv) {

    if (argc < 2) {
        fprintf(stderr, "Not enough arguments!\n");
        return 0;
    }

    if (strcmp(argv[0], "./create_dir") == 0) { create_dir(argv[1]); return 0; }
    if (strcmp(argv[0], "./print_dir") == 0) { print_dir(argv[1]); return 0; }
    if (strcmp(argv[0], "./delete_dir") == 0) { delete_dir(argv[1]); return 0; }
    if (strcmp(argv[0], "./create_file") == 0) { create_file(argv[1]); return 0; }
    if (strcmp(argv[0], "./print_file") == 0) { print_file(argv[1]); return 0; }
    if (strcmp(argv[0], "./delete_file") == 0) { delete_file(argv[1]); return 0; }
    if (strcmp(argv[0], "./create_sym_link") == 0) { create_sym_link(argv[1]); return 0; }
    if (strcmp(argv[0], "./print_sym_link") == 0) { print_sym_link(argv[1]); return 0; }
    if (strcmp(argv[0], "./print_file_sym_link") == 0) { print_file_sym_link(argv[1]); return 0; }
    if (strcmp(argv[0], "./delete_sym_link") == 0) { delete_sym_link(argv[1]); return 0; }
    if (strcmp(argv[0], "./create_hard_link") == 0) { create_hard_link(argv[1]); return 0; }
    if (strcmp(argv[0], "./delete_hard_link") == 0) { delete_hard_link(argv[1]); return 0; }
    if (strcmp(argv[0], "./get_mode") == 0) { get_mode(argv[1]); return 0; }
    if (strcmp(argv[0], "./change_file_mode") == 0) { change_file_mode(argv[1], argv[2]); }
    return 0;
}
```

```
#!/bin/bash
ln second create_dir
ln second print_dir
ln second delete_dir
ln second create_file
ln second print_file
ln second delete_file
ln second create_sym_link
```

```
ln second print_sym_link
ln second print_file_sym_link
ln second delete_sym_link
ln second create_hard_link
ln second delete_hard_link
ln second get_mode
ln second change_file_mode
```

Part 3

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>

#define PAGE_SIZE 0x1000

static void print_page(uint64_t address, uint64_t data) {
    printf("0x%-16lx : pfn %-16lx soft-dirty %ld file-page/shared-anon %ld swapped %ld
present %ld\n", address, data & 0x7fffffffffffff,
                (data >> 55) & 1, (data >> 61) & 1, (data >> 62) & 1, (data >> 63) & 1);
}

int main(int argc, char *argv[]) {
    if (argc != 4) {
        fprintf(stderr, "PID is not specified\n");
        return 0;
    }

    char filename[100];
    errno = 0;
    int pid;
    if (strcmp(argv[1], "self") == 0) {
        pid = getpid();
    } else {
        pid = (int) strtol(argv[1], NULL, 0);
    }
    if (errno) {
        perror("strtol");
        return 1;
    }
    snprintf(filename, sizeof(filename), "/proc/%d/pagemap", pid);

    int fd = open(filename, O_RDONLY);
    if (fd < 0) {
        perror("open");
        return 1;
    }
```

```
    uint64_t start_address = strtoul(argv[2], NULL, 0);
    uint64_t end_address = strtoul(argv[3], NULL, 0);

    for (uint64_t i = start_address; i < end_address; i += PAGE_SIZE) {
        uint64_t data;
        uint64_t index = (i / PAGE_SIZE) * sizeof(data);
        if (pread(fd, &data, sizeof(data), index) != sizeof(data)) {
            perror("pread");
            break;
        }

        print_page(i, data);
    }

    close(fd);
    return 0;
}
```

sudo ./proc1 1 0x0116a000 0x01171000 | grep -v "pfn 0"

```
#include <stdio.h>
#include <unistd.h>

void try_to_open(char* file_name) {
        FILE* file = fopen(file_name, "r");
        if (file == NULL) {
                perror("Impossible to open file\n");
                return;
        } else {
                printf("Good!\n");
        }
        fclose(file);
}

int main(int argc, char** argv) {
        if (argc < 2) {
                printf("File name is not specified\n");
                return -1;
        }

        printf("uid: %d         euid: %d\n", getuid(), geteuid());
        try_to_open(argv[1]);

        if (setuid(getuid()) != 0) {
```

```
                perror("Failed to set uid\n");
                return -1;
        }

        printf("uid: %d         euid: %d\n", getuid(), geteuid());
        try_to_open(argv[1]);

        return 0;
}
```

```
#include <string.h>
#include <stdio.h>
#include <sys/mman.h>
#include <unistd.h>

//#define MAX_POOLS 1000
//#define BUF_SIZE 104888320

const int MAX_POOLS = 1000;
const long int BUF_SIZE = 104890368;

static char* BUFFER;
static unsigned long UNUSED = BUF_SIZE;
static char* pools[MAX_POOLS];
static unsigned int POOLS_COUNT = 1;
static unsigned int pools_size[MAX_POOLS];

static char* blocks[MAX_POOLS];
static unsigned int BLOCKS_COUNT = 0;
static unsigned int block_size[MAX_POOLS];
static unsigned int RIGHT_BLOCK;

static int ERROR = 0;

#define NO_MEMORY 1
#define BLOCK_NOT_FOUND 2


void defrag(void) {
        char* p = BUFFER;
        char* t;
        char* tmp;
```

```
	for (unsigned long int i = 0; i < RIGHT_BLOCK; ++i) {
		printf("iter %d\n", i);
		t = blocks[i];
		printf("%p %p\n", blocks[i], BUFFER);
		if (t == BUFFER) {
			printf("%d %d\n", block_size[i], BUF_SIZE);
			if (block_size[i] == BUF_SIZE) return;
			p = (char* )(blocks[i] + block_size[i] + 1);
			continue;
		}

		printf("1\n");
		tmp = p;
		t = blocks[i];
		printf("2\n");
		for (unsigned long int k = 0; k < block_size[i]; ++k) {
			printf("%d %p\n", k, p);
			*p = *t;
			p++;
			t++;
		}
		printf("3\n");
		blocks[i] = tmp;
	}

	POOLS_COUNT = 1;
	pools[0] = p;
	UNUSED = BUF_SIZE - (unsigned long)(p - BUFFER);
	pools_size[0] = UNUSED;
	RIGHT_BLOCK = 0;
	return;
}

void alloc_init(void) {
	size_t pagesize = getpagesize();
	size_t size = (BUF_SIZE % pagesize == 0) ? (BUF_SIZE) : (BUF_SIZE
+ BUF_SIZE - BUF_SIZE % pagesize);
	printf("size %d pagesize %d\n", size, pagesize);
	BUFFER = (char* )mmap(0, BUF_SIZE, PROT_READ |
PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);
	if (BUFFER == MAP_FAILED) {
		perror("Mmap error");
		_exit(-1);
	}
	pools[0] = BUFFER;
```

```
	printf("init pools %p buffer %p\n", pools[0], BUFFER);
	pools_size[0] = UNUSED;
}

char* my_malloc(unsigned long size) {
	char* p;

	printf("%d %p\n", UNUSED, BUFFER);
	if(size > UNUSED) {
		printf("defrag\n");
		defrag();
	}
	if (size > UNUSED) {
		printf("no memory\n");
		ERROR = NO_MEMORY;
		return 0;
	}

	printf("POOLS_COUNT: %d size: %d\n", POOLS_COUNT,
pools_size[0]);
	p = 0;
	unsigned long int k;
	for (unsigned long int i = 0; i < POOLS_COUNT; ++i) {
		if (size <= pools_size[i]) {
			p = pools[i];
			k = i;
			printf("found %d %p\n", i, pools[i]);
			break;
		}
	}

	if (p == 0) {
		printf("no memory\n");
		ERROR = NO_MEMORY;
		return 0;
	}

	blocks[BLOCKS_COUNT] = p;
	block_size[BLOCKS_COUNT] = size;
	++BLOCKS_COUNT;
	++RIGHT_BLOCK;
	pools[k] = (char*)(p + size + 1);
	pools_size[k] = pools_size[k]-size;

	UNUSED -= size;
```

```
	return p;
}


int my_free(char* block) {
	char* p = 0;
	unsigned int k;
	for (unsigned long int i = 0; i < RIGHT_BLOCK; ++i) {
		if (block == blocks[i]) {
			p = blocks[i];
			k = i;
			break;
		}
	}

	if (p == 0) {
		ERROR = BLOCK_NOT_FOUND;
		return BLOCK_NOT_FOUND;
	}

	blocks[k] = 0;
	--BLOCKS_COUNT;
	pools[POOLS_COUNT] = block;
	pools_size[POOLS_COUNT] = block_size[k];
	++POOLS_COUNT;
	UNUSED += block_size[k];
	return 0;
}

void print_status(char *str) {
	sprintf(str,"Status:\nAvailable: %ld of %ld bytes\nCount of blocks:
%d\n", UNUSED, BUF_SIZE, BLOCKS_COUNT);
}

int check_error(void) {
	int error = ERROR;
	ERROR = 0;
	return error;
}

void print_error(int error_code) {
	if (error_code == 1) {
		perror("Not enough memory");
	}
	if (error_code == 2) {
```

```
            perror("Invalid pointer for free");
        }
}

int main() {
        alloc_init();
        printf("get buffer %p\n", BUFFER);

        char* array1 = my_malloc(104890368);
        printf("pointer: %p\n", array1);
        int error_code = check_error();
        printf("error code: %d\n", error_code);
        if (error_code != 0) {
                print_error(error_code);
        }

        char* array2 = my_malloc(100);
        error_code = check_error();
        printf("error code: %d\n", error_code);
        if (error_code != 0) {
                print_error(error_code);
        }

        my_free(array1);
        error_code = check_error();
        if (error_code != 0) {
                print_error(error_code);
        }
        return 0;
}
```

```
#include <string.h>
#include <stdio.h>
#include <sys/mman.h>
#include <unistd.h>

#define NO_MEMORY 1
#define BLOCK_NOT_FOUND 2

struct heap {
        static const int MAX_POOLS = 1000;
        static const long int BUF_SIZE = 104890368;
```

```
	char* BUFFER;
	unsigned long UNUSED = BUF_SIZE;
	char* pools[MAX_POOLS];
	unsigned int POOLS_COUNT = 1;
	unsigned int pools_size[MAX_POOLS];

	char* blocks[MAX_POOLS];
	unsigned int BLOCKS_COUNT = 0;
	unsigned int block_size[MAX_POOLS];
	unsigned int RIGHT_BLOCK;

	int ERROR = 0;
};

struct heap my_heap;


void defrag(void) {
	char* p = my_heap.BUFFER;
	char* t;
	char* tmp;

	for (unsigned long int i = 0; i < my_heap.RIGHT_BLOCK; ++i) {
		t = my_heap.blocks[i];
		if (t == my_heap.BUFFER) {
			if (my_heap.block_size[i] == my_heap.BUF_SIZE) return;
			p = (char* )(my_heap.blocks[i] + my_heap.block_size[i] + 1);
			continue;
		}

		tmp = p;
		t = my_heap.blocks[i];
		for (unsigned long int k = 0; k < my_heap.block_size[i]; ++k) {
			*p = *t;
			p++;
			t++;
		}
		my_heap.blocks[i] = tmp;
	}

	my_heap.POOLS_COUNT = 1;
	my_heap.pools[0] = p;
	my_heap.UNUSED = my_heap.BUF_SIZE - (unsigned long)(p -
my_heap.BUFFER);
	my_heap.pools_size[0] = my_heap.UNUSED;
```

```
	my_heap.RIGHT_BLOCK = 0;
	return;
}

void alloc_init(void) {
	size_t pagesize = getpagesize();
	size_t size = (my_heap.BUF_SIZE % pagesize == 0) ?
(my_heap.BUF_SIZE) : (my_heap.BUF_SIZE + my_heap.BUF_SIZE -
my_heap.BUF_SIZE % pagesize);
	my_heap.BUFFER = (char* )mmap(0, my_heap.BUF_SIZE,
PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);
	if (my_heap.BUFFER == MAP_FAILED) {
		printf("Mmap error\n");
		_exit(-1);
	}
	my_heap.pools[0] = my_heap.BUFFER;
	my_heap.pools_size[0] = my_heap.UNUSED;
}

char* my_malloc(unsigned long size) {
	char* p;

	if(size > my_heap.UNUSED) {
		defrag();
	}
	if (size > my_heap.UNUSED) {
		my_heap.ERROR = NO_MEMORY;
		return 0;
	}

	p = 0;
	unsigned long int k;
	for (unsigned long int i = 0; i < my_heap.POOLS_COUNT; ++i) {
		if (size <= my_heap.pools_size[i]) {
			p = my_heap.pools[i];
			k = i;
			break;
		}
	}

	if (p == 0) {
		my_heap.ERROR = NO_MEMORY;
		return 0;
	}
```

```
	my_heap.blocks[my_heap.BLOCKS_COUNT] = p;
	my_heap.block_size[my_heap.BLOCKS_COUNT] = size;
	++(my_heap.BLOCKS_COUNT);
	++(my_heap.RIGHT_BLOCK);
	my_heap.pools[k] = (char*)(p + size + 1);
	my_heap.pools_size[k] = my_heap.pools_size[k]-size;

	my_heap.UNUSED -= size;
	return p;
}


int my_free(char* block) {
	char* p = 0;
	unsigned int k;
	for (unsigned long int i = 0; i < my_heap.RIGHT_BLOCK; ++i) {
		if (block == my_heap.blocks[i]) {
			p = my_heap.blocks[i];
			k = i;
			break;
		}
	}

	if (p == 0) {
		my_heap.ERROR = BLOCK_NOT_FOUND;
		return BLOCK_NOT_FOUND;
	}

	my_heap.blocks[k] = 0;
	--(my_heap.BLOCKS_COUNT);
	my_heap.pools[my_heap.POOLS_COUNT] = block;
	my_heap.pools_size[my_heap.POOLS_COUNT] =
my_heap.block_size[k];
	++(my_heap.POOLS_COUNT);
	my_heap.UNUSED += my_heap.block_size[k];
	return 0;
}

void print_status(char *str) {
	sprintf(str,"Status:\nAvailable: %ld of %ld bytes\nCount of blocks:
%d\n", my_heap.UNUSED, my_heap.BUF_SIZE,
my_heap.BLOCKS_COUNT);
}

int check_error(void) {
```

```
	int error = my_heap.ERROR;
	my_heap.ERROR = 0;
	return error;
}

void print_error(int error_code) {
	if (error_code == 1) {
		printf("Not enough memory\n");
	}
	if (error_code == 2) {
		printf("Invalid pointer for free\n");
	}
}

int main() {
	alloc_init();

	char* array1 = my_malloc(104890368);
	printf("pointer for array1: %p\n", array1);
	int error_code = check_error();
	printf("error code: %d\n", error_code);
	if (error_code != 0) {
		print_error(error_code);
	}

	char* array2 = my_malloc(100);
	printf("pointer for array2: %p\n", array2);
	error_code = check_error();
	printf("error code: %d\n", error_code);
	if (error_code != 0) {
		print_error(error_code);
	}

	my_free(array1);
	error_code = check_error();
	if (error_code != 0) {
		print_error(error_code);
	}
	return 0;
}
```

```
#include <string.h>
#include <stdio.h>
```

```
#include <sys/mman.h>
#include <unistd.h>

const int MAX_POOLS = 1000;
const long int BUF_SIZE = 104890368;

static char* BUFFER;
static unsigned long UNUSED = BUF_SIZE;
static char* pools[MAX_POOLS];
static unsigned int POOLS_COUNT = 1;
static unsigned int pools_size[MAX_POOLS];

static char* blocks[MAX_POOLS];
static unsigned int BLOCKS_COUNT = 0;
static unsigned int block_size[MAX_POOLS];
static unsigned int RIGHT_BLOCK;

static int ERROR = 0;

#define NO_MEMORY 1
#define BLOCK_NOT_FOUND 2


void defrag(void) {
	char* p = BUFFER;
	char* t;
	char* tmp;

	for (unsigned long int i = 0; i < RIGHT_BLOCK; ++i) {
		t = blocks[i];
		if (t == BUFFER) {
			if (block_size[i] == BUF_SIZE) return;
			p = (char* )(blocks[i] + block_size[i] + 1);
			continue;
		}

		tmp = p;
		t = blocks[i];
		for (unsigned long int k = 0; k < block_size[i]; ++k) {
			*p = *t;
			p++;
			t++;
		}
		blocks[i] = tmp;
	}
```

```
	POOLS_COUNT = 1;
	pools[0] = p;
	UNUSED = BUF_SIZE - (unsigned long)(p - BUFFER);
	pools_size[0] = UNUSED;
	RIGHT_BLOCK = 0;
	return;
}

void alloc_init(void) {
	size_t pagesize = getpagesize();
	size_t size = (BUF_SIZE % pagesize == 0) ? (BUF_SIZE) : (BUF_SIZE
+ BUF_SIZE - BUF_SIZE % pagesize);
	BUFFER = (char* )mmap(0, BUF_SIZE, PROT_READ |
PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);
	if (BUFFER == MAP_FAILED) {
		printf("Mmap error\n");
		_exit(-1);
	}
	pools[0] = BUFFER;
	pools_size[0] = UNUSED;
}

char* my_malloc(unsigned long size) {
	char* p;

	if(size > UNUSED) {
		defrag();
	}
	if (size > UNUSED) {
		ERROR = NO_MEMORY;
		return 0;
	}

	p = 0;
	unsigned long int k;
	for (unsigned long int i = 0; i < POOLS_COUNT; ++i) {
		if (size <= pools_size[i]) {
			p = pools[i];
			k = i;
			break;
		}
	}

	if (p == 0) {
```

```
            ERROR = NO_MEMORY;
            return 0;
        }

        blocks[BLOCKS_COUNT] = p;
        block_size[BLOCKS_COUNT] = size;
        ++BLOCKS_COUNT;
        ++RIGHT_BLOCK;
        pools[k] = (char*)(p + size + 1);
        pools_size[k] = pools_size[k]-size;

        UNUSED -= size;
        return p;
}


int my_free(char* block) {
        char* p = 0;
        unsigned int k;
        for (unsigned long int i = 0; i < RIGHT_BLOCK; ++i) {
                if (block == blocks[i]) {
                        p = blocks[i];
                        k = i;
                        break;
                }
        }

        if (p == 0) {
                ERROR = BLOCK_NOT_FOUND;
                return BLOCK_NOT_FOUND;
        }

        blocks[k] = 0;
        --BLOCKS_COUNT;
        pools[POOLS_COUNT] = block;
        pools_size[POOLS_COUNT] = block_size[k];
        ++POOLS_COUNT;
        UNUSED += block_size[k];
        return 0;
}

void print_status(char *str) {
        sprintf(str,"Status:\nAvailable: %ld of %ld bytes\nCount of blocks:
%d\n", UNUSED, BUF_SIZE, BLOCKS_COUNT);
}
```

```
int check_error(void) {
	int error = ERROR;
	ERROR = 0;
	return error;
}

void print_error(int error_code) {
	if (error_code == 1) {
		printf("Not enough memory\n");
	}
	if (error_code == 2) {
		printf("Invalid pointer for free\n");
	}
}

int main() {
	alloc_init();

	char* array1 = my_malloc(104890368);
	printf("pointer for array1: %p\n", array1);
	int error_code = check_error();
	printf("error code: %d\n", error_code);
	if (error_code != 0) {
		print_error(error_code);
	}

	char* array2 = my_malloc(100);
	printf("pointer for array2: %p\n", array2);
	error_code = check_error();
	printf("error code: %d\n", error_code);
	if (error_code != 0) {
		print_error(error_code);
	}

	my_free(array1);
	error_code = check_error();
	if (error_code != 0) {
		print_error(error_code);
	}
	return 0;
}
```

```
#include <string.h>
#include <stdio.h>
#include <sys/mman.h>
#include <unistd.h>
#include <fcntl.h>

#define NO_MEMORY 1
#define BLOCK_NOT_FOUND 2

struct heap {
	static const int MAX_POOLS = 1000;
	static const long int BUF_SIZE = 104857600; //100Mb

	char* BUFFER;
	unsigned long UNUSED = BUF_SIZE;
	char* pools[MAX_POOLS];
	unsigned int POOLS_COUNT = 1;
	unsigned int pools_size[MAX_POOLS];

	char* blocks[MAX_POOLS];
	unsigned int BLOCKS_COUNT = 0;
	unsigned int block_size[MAX_POOLS];
	unsigned int RIGHT_BLOCK;

	int ERROR = 0;
};

struct heap my_heap;


void defrag(void) {
	char* p = my_heap.BUFFER;
	char* t;
	char* tmp;

	for (unsigned long int i = 0; i < my_heap.RIGHT_BLOCK; ++i) {
		t = my_heap.blocks[i];
		if (t == my_heap.BUFFER) {
			if (my_heap.block_size[i] == my_heap.BUF_SIZE) return;
			p = (char* )(my_heap.blocks[i] + my_heap.block_size[i] + 1);
			continue;
		}

		tmp = p;
		t = my_heap.blocks[i];
```

```
            for (unsigned long int k = 0; k < my_heap.block_size[i]; ++k) {
                *p = *t;
                p++;
                t++;
            }
            my_heap.blocks[i] = tmp;
        }

        my_heap.POOLS_COUNT = 1;
        my_heap.pools[0] = p;
        my_heap.UNUSED = my_heap.BUF_SIZE - (unsigned long)(p -
my_heap.BUFFER);
        my_heap.pools_size[0] = my_heap.UNUSED;
        my_heap.RIGHT_BLOCK = 0;
        return;
}

void alloc_init(void) {
        size_t pagesize = getpagesize();

        const char *filepath = "heap.txt";
        int fd = open(filepath, O_RDWR);
        if(fd < 0){
            printf("Failed to open file %s\n", filepath);
            _exit(-1);
        }

        my_heap.BUFFER = (char* )mmap(0, my_heap.BUF_SIZE,
PROT_READ | PROT_WRITE, /*MAP_PRIVATE | MAP_ANONYMOUS*/
MAP_SHARED, fd, 0);
        close(fd);

        if (my_heap.BUFFER == MAP_FAILED) {
            perror("Mmap error\n");
            _exit(-1);
        }
        my_heap.pools[0] = my_heap.BUFFER;
        my_heap.pools_size[0] = my_heap.UNUSED;

        for (int i = 0; i < my_heap.BUF_SIZE; ++i) {
            my_heap.BUFFER[i] = 0;
        }
}

void dectroy() {
```

```
	int result = munmap(my_heap.BUFFER, my_heap.BUF_SIZE);
	if (result != 0) {
		perror("Unmap error\n");
		_exit(-1);
	}
}

char* my_malloc(unsigned long size) {
	char* p;

	if(size > my_heap.UNUSED) {
		defrag();
	}
	if (size > my_heap.UNUSED) {
		my_heap.ERROR = NO_MEMORY;
		return 0;
	}

	p = 0;
	unsigned long int k;
	for (unsigned long int i = 0; i < my_heap.POOLS_COUNT; ++i) {
		if (size <= my_heap.pools_size[i]) {
			p = my_heap.pools[i];
			k = i;
			break;
		}
	}

	if (p == 0) {
		my_heap.ERROR = NO_MEMORY;
		return 0;
	}

	my_heap.blocks[my_heap.BLOCKS_COUNT] = p;
	my_heap.block_size[my_heap.BLOCKS_COUNT] = size;
	++(my_heap.BLOCKS_COUNT);
	++(my_heap.RIGHT_BLOCK);
	my_heap.pools[k] = (char*)(p + size + 1);
	my_heap.pools_size[k] = my_heap.pools_size[k]-size;

	my_heap.UNUSED -= size;
	return p;
}
```

```
int my_free(char* block) {
	char* p = 0;
	unsigned int k;
	for (unsigned long int i = 0; i < my_heap.RIGHT_BLOCK; ++i) {
		if (block == my_heap.blocks[i]) {
			p = my_heap.blocks[i];
			k = i;
			break;
		}
	}

	if (p == 0) {
		my_heap.ERROR = BLOCK_NOT_FOUND;
		return BLOCK_NOT_FOUND;
	}

	my_heap.blocks[k] = 0;
	--(my_heap.BLOCKS_COUNT);
	my_heap.pools[my_heap.POOLS_COUNT] = block;
	my_heap.pools_size[my_heap.POOLS_COUNT] =
my_heap.block_size[k];
	++(my_heap.POOLS_COUNT);
	my_heap.UNUSED += my_heap.block_size[k];
	return 0;
}

void print_status(char *str) {
	sprintf(str,"Status:\nAvailable: %ld of %ld bytes\nCount of blocks:
%d\n", my_heap.UNUSED, my_heap.BUF_SIZE,
my_heap.BLOCKS_COUNT);
}

int check_error(void) {
	int error = my_heap.ERROR;
	my_heap.ERROR = 0;
	return error;
}

void print_error(int error_code) {
	if (error_code == 1) {
		printf("Not enough memory\n");
	}
	if (error_code == 2) {
		printf("Invalid pointer for free\n");
	}
```

```
}

int main() {
	alloc_init();

	char* array1 = my_malloc(104857600);
	printf("pointer for array1: %p\n", array1);
	int error_code = check_error();
	printf("error code: %d\n", error_code);
	if (error_code != 0) {
		print_error(error_code);
	}

	for (int i = 0; i < 1000; ++i) {
		array1[i] = '5';
	}
	for (int i = 0; i < 10; ++i ) {
		printf("%c ", array1[i]);
	}
	printf("\n");
/*
	char* array2 = my_malloc(100);
	printf("pointer for array2: %p\n", array2);
	error_code = check_error();
	printf("error code: %d\n", error_code);
	if (error_code != 0) {
		print_error(error_code);
	}
*/
	my_free(array1);
	error_code = check_error();
	if (error_code != 0) {
		print_error(error_code);
	}
	return 0;
}
```

```
#include <string.h>
#include <stdio.h>
#include <sys/mman.h>
#include <unistd.h>
#include <fcntl.h>
```

```
#define NO_MEMORY 1
#define BLOCK_NOT_FOUND 2
#define NULL_POINTER 3

struct heap {
	static const unsigned long int MAX_POOLS = 1000;
	static const unsigned long int BUF_SIZE = 104857600; //100Mb

	char* BUFFER;
	unsigned long int UNUSED = BUF_SIZE;
	char* pools[MAX_POOLS];
	unsigned long int POOLS_COUNT = 1;
	unsigned long int pools_size[MAX_POOLS];

	char* blocks[MAX_POOLS];
	unsigned long int BLOCKS_COUNT = 0;
	unsigned long int RIGHT_BLOCK;

	int ERROR = 0;
};

struct heap my_heap;


unsigned long int read_size(char* data) {
	unsigned long int size = 0;
	char* ptr = (char*)(&size);
	*(ptr + 7) = data[0];
	*(ptr + 6) = data[1];
	*(ptr + 5) = data[2];
	*(ptr + 4) = data[3];
	*(ptr + 3) = data[4];
	*(ptr + 2) = data[5];
	*(ptr + 1) = data[6];
	*ptr = data[7];
	return size;
}

void write_size(char* p, unsigned long int size) {
	p[0] = (char) (size >> 56);
	p[1] = (char) (size >> 48);
	p[2] = (char) (size >> 40);
	p[3] = (char) (size >> 32);
	p[4] = (char) (size >> 24);
```

```
	p[5] = (char) (size >> 16);
	p[6] = (char) (size >> 8);
	p[7] = (char) (size);
}


void defrag(void) {
	char* p = my_heap.BUFFER;
	char* t;
	char* tmp;

	for (unsigned long int i = 0; i < my_heap.RIGHT_BLOCK; ++i) {
		t = my_heap.blocks[i];
		unsigned long int block_size = *((unsigned long int
*)my_heap.blocks[i]);
		if (t == my_heap.BUFFER) {
			if (block_size == my_heap.BUF_SIZE) return;
			p = (char* )(my_heap.blocks[i] + block_size + 1);
			continue;
		}

		tmp = p;
		t = my_heap.blocks[i];
		for (unsigned long int k = 0; k < block_size; ++k) {
			*p = *t;
			p++;
			t++;
		}
		my_heap.blocks[i] = tmp;
	}

	my_heap.POOLS_COUNT = 1;
	my_heap.pools[0] = p;
	my_heap.UNUSED = my_heap.BUF_SIZE - (unsigned long int)(p -
my_heap.BUFFER);
	my_heap.pools_size[0] = my_heap.UNUSED;
	my_heap.RIGHT_BLOCK = 0;
	return;
}

void alloc_init(void) {
	size_t pagesize = getpagesize();

	const char *filepath = "heap.txt";
	int fd = open(filepath, O_RDWR);
```

```
	if(fd < 0){
		printf("Failed to open file %s\n", filepath);
		_exit(-1);
	}

	my_heap.BUFFER = (char* )mmap(0, my_heap.BUF_SIZE,
PROT_READ | PROT_WRITE, /*MAP_PRIVATE | MAP_ANONYMOUS*/
MAP_SHARED, fd, 0);
	close(fd);

	if (my_heap.BUFFER == MAP_FAILED) {
		perror("Mmap error\n");
		_exit(-1);
	}
	my_heap.pools[0] = my_heap.BUFFER;
	my_heap.pools_size[0] = my_heap.UNUSED;

	for (int i = 0; i < my_heap.BUF_SIZE; ++i) {
		my_heap.BUFFER[i] = 0;
	}
}

void dectroy() {
	int result = munmap(my_heap.BUFFER, my_heap.BUF_SIZE);
	if (result != 0) {
		perror("Unmap error\n");
		_exit(-1);
	}
}

char* my_malloc(unsigned long int size) {
	char* p;

	if(size + sizeof(unsigned long int) > my_heap.UNUSED) {
		defrag();
	}
	if (size + sizeof(unsigned long int) > my_heap.UNUSED) {
		my_heap.ERROR = NO_MEMORY;
		return 0;
	}

	p = 0;
	unsigned long int k;
	for (unsigned long int i = 0; i < my_heap.POOLS_COUNT; ++i) {
		if (size + sizeof(unsigned long int) <= my_heap.pools_size[i]) {
```

```
                p = my_heap.pools[i];
                k = i;
                break;
            }
        }

        if (p == 0) {
            my_heap.ERROR = NO_MEMORY;
            return 0;
        }

        my_heap.blocks[my_heap.BLOCKS_COUNT] = p;
        write_size(p, size);

        ++(my_heap.BLOCKS_COUNT);
        ++(my_heap.RIGHT_BLOCK);
        my_heap.pools[k] = (char*)(p + size + 1);
        my_heap.pools_size[k] = my_heap.pools_size[k]-size;

        my_heap.UNUSED -= size;
        return p + sizeof(unsigned long int);
}


int my_free(char* block) {
        if (block == 0) {
            my_heap.ERROR = NULL_POINTER;
            return NULL_POINTER;
        }

        block -= sizeof(unsigned long int);

        unsigned long int block_size = read_size(block);
        printf("size: %lu\n", block_size);

        char* p = 0;
        unsigned long int k;
        for (unsigned long int i = 0; i < my_heap.RIGHT_BLOCK; ++i) {
            if (block == my_heap.blocks[i]) {
                p = my_heap.blocks[i];
                k = i;
                break;
            }
        }
```

```
	if (p == 0) {
		my_heap.ERROR = BLOCK_NOT_FOUND;
		return BLOCK_NOT_FOUND;
	}

	my_heap.blocks[k] = 0;
	--(my_heap.BLOCKS_COUNT);
	my_heap.pools[my_heap.POOLS_COUNT] = block;
	my_heap.pools_size[my_heap.POOLS_COUNT] = block_size;
	my_heap.pools_size[my_heap.POOLS_COUNT] = block_size;
	++(my_heap.POOLS_COUNT);
	my_heap.UNUSED += block_size;
	return 0;
}

void print_status(char *str) {
	sprintf(str,"Status:\nAvailable: %lu of %lu bytes\nCount of blocks:
%lu\n", my_heap.UNUSED, my_heap.BUF_SIZE,
my_heap.BLOCKS_COUNT);
}

int check_error(void) {
	int error = my_heap.ERROR;
	my_heap.ERROR = 0;
	return error;
}

void print_error(int error_code) {
	if (error_code == 1) {
		printf("Not enough memory\n");
	}
	if (error_code == 2) {
		printf("Invalid pointer for free\n");
	}
}

int main() {
	alloc_init();

	char* array1 = my_malloc(104857592);
	printf("pointer for array1: %p\n", array1);
	int error_code = check_error();
	printf("error code: %d\n", error_code);
	if (error_code != 0) {
		print_error(error_code);
```

```
            return -1;
        }

        for (int i = 0; i < 100; ++i) {
            array1[i] = '3';
        }
        for (int i = 0; i < 10; ++i ) {
            printf("%c ", array1[i]);
        }
        printf("\n");
/*
        char* array2 = my_malloc(100);
        printf("pointer for array2: %p\n", array2);
        error_code = check_error();
        printf("error code: %d\n", error_code);
        if (error_code != 0) {
            print_error(error_code);
        }
*/
        my_free(array1);
        error_code = check_error();
        if (error_code != 0) {
            print_error(error_code);
        }
        return 0;
}
```

```
#include <string.h>
#include <stdio.h>
#include <sys/mman.h>
#include <unistd.h>
#include <fcntl.h>

#define NO_MEMORY 1
#define BLOCK_NOT_FOUND 2
#define NULL_POINTER 3

struct heap {
        static const unsigned long int MAX_POOLS = 1000;
        static const unsigned long int BUF_SIZE = 104857600; //100Mb

        char* BUFFER;
        unsigned long int UNUSED = BUF_SIZE;
```

```
	char* pools[MAX_POOLS];
	unsigned long int POOLS_COUNT = 1;

	char* blocks[MAX_POOLS];
	unsigned long int BLOCKS_COUNT = 0;
	unsigned long int RIGHT_BLOCK;

	int ERROR = 0;
};

struct heap my_heap;


unsigned long int read_size(char* data) {
	unsigned long int size = 0;
	char* ptr = (char*)(&size);
	*(ptr + 7) = data[0];
	*(ptr + 6) = data[1];
	*(ptr + 5) = data[2];
	*(ptr + 4) = data[3];
	*(ptr + 3) = data[4];
	*(ptr + 2) = data[5];
	*(ptr + 1) = data[6];
	*ptr = data[7];
	return size;
}

void write_size(char* p, unsigned long int size) {
	p[0] = (char) (size >> 56);
	p[1] = (char) (size >> 48);
	p[2] = (char) (size >> 40);
	p[3] = (char) (size >> 32);
	p[4] = (char) (size >> 24);
	p[5] = (char) (size >> 16);
	p[6] = (char) (size >> 8);
	p[7] = (char) (size);
}


void defrag(void) {
	char* p = my_heap.BUFFER;
	char* t;
	char* tmp;

	for (unsigned long int i = 0; i < my_heap.RIGHT_BLOCK; ++i) {
```

```
            t = my_heap.blocks[i];
            unsigned long int block_size = read_size(my_heap.blocks[i]);
            if (t == my_heap.BUFFER) {
                  if (block_size == (my_heap.BUF_SIZE - sizeof(unsigned long
int))) {
                        printf("return\n");
                        return;
                  }
                  p = (char* )(my_heap.blocks[i] + block_size +
sizeof(unsigned long int) + 1);
                  continue;
            }

            tmp = p;
            t = my_heap.blocks[i];
            for (unsigned long int k = 0; k < block_size; ++k) {
                  *p = *t;
                  p++;
                  t++;
            }
            my_heap.blocks[i] = tmp;
      }

      my_heap.POOLS_COUNT = 1;
      my_heap.pools[0] = p;
      my_heap.UNUSED = my_heap.BUF_SIZE - (unsigned long int)(p -
my_heap.BUFFER);
      write_size(my_heap.pools[0], my_heap.UNUSED);
      my_heap.RIGHT_BLOCK = 0;
      return;
}

void init(void) {
      size_t pagesize = getpagesize();

      const char *filepath = "heap.txt";
      int fd = open(filepath, O_RDWR);
      if(fd < 0){
            printf("Failed to open file %s\n", filepath);
            _exit(-1);
      }

      my_heap.BUFFER = (char* )mmap(0, my_heap.BUF_SIZE,
PROT_READ | PROT_WRITE, /*MAP_PRIVATE | MAP_ANONYMOUS*/
MAP_SHARED, fd, 0);
```

```
	close(fd);

	if (my_heap.BUFFER == MAP_FAILED) {
		perror("Mmap error\n");
		_exit(-1);
	}

	for (int i = 0; i < my_heap.BUF_SIZE; ++i) {
		my_heap.BUFFER[i] = 0;
	}

	my_heap.pools[0] = my_heap.BUFFER;
	write_size(my_heap.pools[0], my_heap.UNUSED);
}

void destroy() {
	int result = munmap(my_heap.BUFFER, my_heap.BUF_SIZE);
	if (result != 0) {
		perror("Unmap error\n");
		_exit(-1);
	}
}

char* my_malloc(unsigned long int size) {
	char* p;

	if(size + sizeof(unsigned long int) > my_heap.UNUSED) {
		defrag();
	}
	if (size + sizeof(unsigned long int) > my_heap.UNUSED) {
		my_heap.ERROR = NO_MEMORY;
		return 0;
	}

	p = 0;
	unsigned long int k;
	unsigned long int s;
	for (unsigned long int i = 0; i < my_heap.POOLS_COUNT; ++i) {
		unsigned long int pool_size = read_size(my_heap.pools[i]);
		printf("free size for %lu %lu\n", i, pool_size);
		if (size + sizeof(unsigned long int) <= pool_size) {
			p = my_heap.pools[i];
			k = i;
			s = pool_size;
			break;
```

```
            }
        }

        if (p == 0) {
            my_heap.ERROR = NO_MEMORY;
            return 0;
        }

        my_heap.blocks[my_heap.BLOCKS_COUNT] = p;
        write_size(p, size);

        ++(my_heap.BLOCKS_COUNT);
        ++(my_heap.RIGHT_BLOCK);
        if (s - size <= 8) {
            --(my_heap.POOLS_COUNT);
            my_heap.pools[k] = 0;
        } else {
            my_heap.pools[k] = (char*)(p + size + 1);
            printf("5 %lu %lu\n", k, s);
            write_size(my_heap.pools[k], s - size-sizeof(unsigned long int));
        }
        my_heap.UNUSED -= (size + sizeof(unsigned long int));
        return p + sizeof(unsigned long int);
}


int my_free(char* block) {
        if (block == 0) {
            my_heap.ERROR = NULL_POINTER;
            return NULL_POINTER;
        }

        block -= sizeof(unsigned long int);

        unsigned long int block_size = read_size(block);
        printf("size: %lu\n", block_size);

        char* p = 0;
        unsigned long int k;
        for (unsigned long int i = 0; i < my_heap.RIGHT_BLOCK; ++i) {
            if (block == my_heap.blocks[i]) {
                p = my_heap.blocks[i];
                k = i;
                break;
            }
```

```
	}

	if (p == 0) {
		my_heap.ERROR = BLOCK_NOT_FOUND;
		return BLOCK_NOT_FOUND;
	}

	my_heap.blocks[k] = 0;
	--(my_heap.BLOCKS_COUNT);
	my_heap.pools[my_heap.POOLS_COUNT] = block;
	write_size(my_heap.pools[my_heap.POOLS_COUNT], block_size +
sizeof(unsigned long int));
	++(my_heap.POOLS_COUNT);
	my_heap.UNUSED += (block_size + sizeof(unsigned long int));
	return 0;
}

void print_status(char *str) {
	sprintf(str,"Status:\nAvailable: %lu of %lu bytes\nCount of blocks:
%lu\n", my_heap.UNUSED, my_heap.BUF_SIZE,
my_heap.BLOCKS_COUNT);
}

int check_error(void) {
	int error = my_heap.ERROR;
	my_heap.ERROR = 0;
	return error;
}

void print_error(int error_code) {
	if (error_code == 1) {
		printf("Not enough memory\n");
	}
	else if (error_code == 2) {
		printf("Invalid pointer for free\n");
	}
	else if (error_code == 3) {
		printf("Null pointer error\n");
	}
}

int main() {
	init();

	char* array1 = my_malloc(104857492);
```

```
	printf("pointer for array1: %p\n", array1);
	int error_code = check_error();
	printf("error code: %d\n", error_code);
	if (error_code != 0) {
		print_error(error_code);
		return -1;
	}

	for (int i = 0; i < 100; ++i) {
		array1[i] = '3';
	}
	for (int i = 0; i < 10; ++i ) {
		printf("%c ", array1[i]);
	}
	printf("\n");

	char* array2 = my_malloc(92);
	printf("pointer for array2: %p\n", array2);
	error_code = check_error();
	printf("error code: %d\n", error_code);
	if (error_code != 0) {
		print_error(error_code);
	}

	my_free(array1);
	error_code = check_error();
	if (error_code != 0) {
		print_error(error_code);
	}
	my_free(array2);
	error_code = check_error();
	if (error_code != 0) {
		print_error(error_code);
	}
	destroy();
	return 0;
}
```