Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

Лабораторная работа №4 по курсу

«Операционные системы»

Группа: М8О-213Б-23

Студент: Чувилов А.А.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 26.12.24

Москва, 2024

Постановка задачи

Вариант 7.

Требуется создать две динамические библиотеки, реализующие два аллокатора: Списки свободных блоков (наиболее подходящее) и блоки по 2^n.

Общий метод и алгоритм решения

Использованные системные вызовы:

1. ***int munmap(void addr, size_t length); -** Удаляет все отображения из заданной области памяти.
2. ***int dlclose(void handle);** - Закрывает динамическую библиотеку, открытую с помощью dlopen, и освобождает ресурсы, связанные с этим дескриптором.
3. ****void dlopen(const char filename, int flag);** - Открывает динамическую библиотеку и возвращает дескриптор для последующего использования.
4. ****void mmap(void addr, size_t length, int prot, int flags, int fd, off_t offset);** – создает новое отображение памяти или изменяет существующее**.**
5. **int write(int _Filehandle, const void *_Buf, unsigned int _MaxCharCount) –** выводит информацию в файл с указанным дескриптором.

Описание программы

1. main.c

Данный код реализует аллокатор памяти, который позволяет управлять выделением и освобождением памяти в пределах заранее выделенного блока. Он использует структуру данных BlockHeader для описания свободных и занятых блоков памяти, а также связный список для отслеживания свободных блоков. Аллокатор поддерживает следующие функции: создание аллокатора (allocator_create), выделение блока памяти заданного размера с выравниванием (allocator_alloc), освобождение блока памяти с объединением смежных свободных блоков (allocator_free) и уничтожение аллокатора с освобождением всех ресурсов (allocator_destroy). Основная задача аллокатора — эффективно управлять памятью внутри заданного блока, минимизируя фрагментацию и обеспечивая возможность многократного использования.

2. list_alocator.c

Данный код представляет собой реализацию простого аллокатора памяти на основе списка свободных блоков (free list). Аллокатор выделяет память внутри заданного заранее блока памяти, разбивая её на блоки, которые отслеживаются с помощью структуры BlockHeader. Функция allocator_create инициализирует аллокатор, определяя начальную структуру свободного блока. Функция allocator_alloc ищет наиболее подходящий (best-fit) свободный блок для выделения памяти, разбивает его при необходимости и помечает как занятый. Функция allocator_free освобождает указанный блок памяти, возвращая его в список свободных и объединяя смежные блоки для минимизации фрагментации. Наконец, функция allocator_destroy освобождает весь выделенный блок памяти. Основное назначение данного аллокатора — эффективное управление памятью в ограниченном пространстве, минимизация фрагментации и поддержка динамического выделения памяти.

3. buddy.c

Данный код реализует аллокатор памяти на основе системы buddy allocation. Аллокатор разбивает выделенную память на блоки степеней двойки, которые отслеживаются в нескольких списках свободных блоков, упорядоченных по размеру. Функция allocator_create инициализирует аллокатор, организуя память в систему buddy-блоков. Функция allocator_alloc выделяет блок памяти подходящего размера, при необходимости разделяя более крупные блоки (split). Функция allocator_free освобождает указанный блок и объединяет (coalesce) его с соседними блоками, если те также свободны, чтобы минимизировать фрагментацию. Функция allocator_destroy освобождает ресурсы, связанные с аллокатором. Такая структура обеспечивает быстрое выделение и освобождение памяти с минимальной фрагментацией, благодаря гибкости системы buddy allocation.

Код программы

main.c

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

#define MEMORY_SIZE 1024 * 1024

typedef struct Allocator {
    void* memory_start;
    size_t memory_size;
    void* free_list;
} Allocator;

Allocator* allocator_create(void* const memory, const size_t size);
void allocator_destroy(Allocator* const allocator);
void* allocator_alloc(Allocator* const allocator, const size_t size);
void allocator_free(Allocator* const allocator, void* const memory);

void my_write(const char* message) {
    write(STDERR_FILENO, message, strlen(message));
}


void my_write_hex(void* ptr) {
    char buffer[64];
    unsigned long addr = (unsigned long)ptr;
    size_t i;
    for (i = 0; i < sizeof(buffer) - 1 && addr; ++i) {
        unsigned char byte = addr & 0xF;
        buffer[i] = (byte < 10) ? '0' + byte : 'a' + (byte - 10);
        addr >>= 4;
    }
    buffer[i] = '\0';
    my_write(buffer);
}

int main(int argc, char* argv[]) {
    if (argc < 2) {
        my_write("Usage: <path_to_allocator_library>\n");
        return 1;
    }

    void* handle = dlopen(argv[1], RTLD_LAZY);
    if (!handle) {
        my_write("Failed to load library: ");
        my_write(dlerror());
        my_write("\n");
```

```
    return 1;
}

Allocator* (*allocator_create)(void*, size_t) = dlsym(handle, "allocator_create");
void (*allocator_destroy)(Allocator*) = dlsym(handle, "allocator_destroy");
void* (*allocator_alloc)(Allocator*, size_t) = dlsym(handle, "allocator_alloc");
void (*allocator_free)(Allocator*, void*) = dlsym(handle, "allocator_free");

char* error;
if ((error = dlerror()) != NULL) {
    my_write("Error resolving symbols: ");
    my_write(error);
    my_write("\n");
    dlclose(handle);
    return 1;
}

void* memory = mmap(NULL, MEMORY_SIZE, PROT_READ | PROT_WRITE, MAP_PRIVATE |
MAP_ANONYMOUS, -1, 0);
if (memory == MAP_FAILED) {
    my_write("mmap failed\n");
    dlclose(handle);
    return 1;
}

Allocator* allocator = allocator_create(memory, MEMORY_SIZE);
if (!allocator) {
    my_write("Failed to create allocator\n");
    munmap(memory, MEMORY_SIZE);
    dlclose(handle);
    return 1;
}

void* block = allocator_alloc(allocator, 128);
if (block) {
    my_write("Allocated block at ");
    my_write_hex(block);
    my_write("\n");
} else {
    my_write("Failed to allocate block\n");
}

allocator_free(allocator, block);
my_write("Freed block\n");

allocator_destroy(allocator);
munmap(memory, MEMORY_SIZE);
dlclose(handle);

return 0;
```

}

Buddy.c

```
#include <math.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <unistd.h>

#define MIN_BLOCK_SIZE 16

int log2s(size_t n) {
   if (n == 0) {
      return -1;
   }
   int result = 0;
   while (n > 1) {
      n >>= 1;
      result++;
   }
   return result;
}

typedef struct BlockHeader {
   struct BlockHeader *next;
} BlockHeader;

typedef struct Allocator {
   BlockHeader **free_lists;
   size_t num_lists;
   void *base_addr;
   size_t total_size;
} Allocator;

Allocator *allocator_create(void *memory, size_t size) {
   if (!memory || size < sizeof(Allocator)) {
      return NULL;
   }
   Allocator *allocator = (Allocator *)memory;
   allocator->base_addr = memory;
   allocator->total_size = size;

   size_t max_block_size = size;

   allocator->num_lists = (size_t)(log2s(max_block_size) - log2s(MIN_BLOCK_SIZE) + 1);
   allocator->free_lists =
      (BlockHeader **)((char *)memory + sizeof(Allocator));

   for (size_t i = 0; i < allocator->num_lists; i++) {
      allocator->free_lists[i] = NULL;
   }

   void *current_block = (char *)memory + sizeof(Allocator) +
                         allocator->num_lists * sizeof(BlockHeader *);
```

```
    size_t remaining_size =
        size - sizeof(Allocator) - allocator->num_lists * sizeof(BlockHeader *);

    size_t block_size = MIN_BLOCK_SIZE;
    while (remaining_size >= block_size) {
        BlockHeader *header = (BlockHeader *)current_block;
        size_t index = log2s(block_size) - log2s(MIN_BLOCK_SIZE);
        header->next = allocator->free_lists[index];
        allocator->free_lists[index] = header;

        current_block = (char *)current_block + block_size;
        remaining_size -= block_size;
        block_size <<= 1;
    }
    return allocator;
}

void *allocator_alloc(Allocator *allocator, size_t size) {
    if (!allocator || size == 0) {
        return NULL;
    }

    size_t adjusted_size = size < MIN_BLOCK_SIZE ? MIN_BLOCK_SIZE : size;
    size_t index = log2s(adjusted_size) - log2s(MIN_BLOCK_SIZE);

    if (index >= allocator->num_lists) {
        return NULL;
    }

    for (size_t i = index; i < allocator->num_lists; i++) {
        if (allocator->free_lists[i] != NULL) {
            BlockHeader *block = allocator->free_lists[i];
            allocator->free_lists[i] = block->next;

            size_t block_size = MIN_BLOCK_SIZE << i;
            while (i > index) {
                i--;
                block_size >>= 1;

                BlockHeader *split_block =
                    (BlockHeader *)((char *)block + block_size);
                split_block->next = allocator->free_lists[i];
                allocator->free_lists[i] = split_block;
            }

            return (void *)((char *)block + sizeof(BlockHeader));
        }
    }

    return NULL;
}

void allocator_free(Allocator *allocator, void *ptr) {
    if (!allocator || !ptr) {
        return;
    }
```

```
    BlockHeader *block = (BlockHeader *)((char *)ptr - sizeof(BlockHeader));
    size_t block_offset = (char *)block - (char *)allocator->base_addr;
    size_t block_size = MIN_BLOCK_SIZE;

    while (block_offset % (block_size * 2) == 0 && block_size < allocator->total_size) {
        size_t buddy_offset = block_offset ^ block_size;
        BlockHeader *buddy = (BlockHeader *)((char *)allocator->base_addr + buddy_offset);
        size_t buddy_index = log2s(block_size) - log2s(MIN_BLOCK_SIZE);


        BlockHeader *prev = NULL;
        BlockHeader *curr = allocator->free_lists[buddy_index];
        while (curr) {
            if (curr == buddy) {
                if (prev) {
                    prev->next = curr->next;
                } else {
                    allocator->free_lists[buddy_index] = curr->next;
                }
                block_offset &= ~(block_size * 2 - 1);
                block = (BlockHeader *)((char *)allocator->base_addr + block_offset);
                block_size *= 2;
                goto continue_coalescing;
            }
            prev = curr;
            curr = curr->next;
        }
        break;

    continue_coalescing:;
    }

    size_t index = log2s(block_size) - log2s(MIN_BLOCK_SIZE);
    block->next = allocator->free_lists[index];
    allocator->free_lists[index] = block;
}

void allocator_destroy(Allocator *allocator) {
    if (allocator) {
        munmap(allocator->base_addr, allocator->total_size);
    }
}
```

list_allocator.c

```
#include <math.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <unistd.h>

#define MIN_BLOCK_SIZE 16

typedef struct BlockHeader {
```

```
    size_t size;
    struct BlockHeader *next;
    bool is_free;
} BlockHeader;

typedef struct Allocator {
    BlockHeader *free_list;
    void *memory_start;
    size_t total_size;
    void *base_addr;
} Allocator;

Allocator *allocator_create(void *memory, size_t size) {
    if (!memory || size < sizeof(Allocator)) {
        return NULL;
    }

    Allocator *allocator = (Allocator *)memory;
    allocator->base_addr = memory;
    allocator->memory_start = (char *)memory + sizeof(Allocator);
    allocator->total_size = size - sizeof(Allocator);
    allocator->free_list = (BlockHeader *)allocator->memory_start;

    allocator->free_list->size = allocator->total_size - sizeof(BlockHeader);
    allocator->free_list->next = NULL;
    allocator->free_list->is_free = true;

    return allocator;
}

void *allocator_alloc(Allocator *allocator, size_t size) {
    if (!allocator || size == 0) {
        return NULL;
    }

    size = (size + MIN_BLOCK_SIZE - 1) / MIN_BLOCK_SIZE *
           MIN_BLOCK_SIZE;

    BlockHeader *best_fit = NULL;
    BlockHeader *prev_best = NULL;
    BlockHeader *current = allocator->free_list;
    BlockHeader *prev = NULL;

    while (current) {
        if (current->is_free && current->size >= size) {
            if (best_fit == NULL || current->size < best_fit->size) {
                best_fit = current;
                prev_best = prev;
            }
        }
        prev = current;
        current = current->next;
    }

    if (best_fit) {
        size_t remaining_size = best_fit->size - size;
        if (remaining_size >= sizeof(BlockHeader) + MIN_BLOCK_SIZE) {
```

```
            BlockHeader *new_block =
                (BlockHeader *)((char *)best_fit + sizeof(BlockHeader) + size);
            new_block->size = remaining_size - sizeof(BlockHeader);
            new_block->is_free = true;
            new_block->next = best_fit->next;

            best_fit->next = new_block;
            best_fit->size = size;
        }

        best_fit->is_free = false;
        if (prev_best == NULL) {
            allocator->free_list = best_fit->next;
        } else {
            prev_best->next = best_fit->next;
        }

        return (void *)((char *)best_fit + sizeof(BlockHeader));
    }

    return NULL;
}

void allocator_free(Allocator *allocator, void *ptr) {
    if (!allocator || !ptr) {
        return;
    }

    BlockHeader *header = (BlockHeader *)((char *)ptr - sizeof(BlockHeader));
    if (!header) return;
    header->is_free = true;

    header->next = allocator->free_list;
    allocator->free_list = header;

    BlockHeader *current = allocator->free_list;
    BlockHeader *prev = NULL;

    while (current && current->next) {
        BlockHeader *next = current->next;
        if (((char *)current + sizeof(BlockHeader) + current->size) ==
            (char *)next) {
            current->size += next->size + sizeof(BlockHeader);
            current->next = next->next;
            continue;
        }
        if (prev && ((char *)prev + sizeof(BlockHeader) + prev->size) ==
                    (char *)current) {
            prev->size += current->size + sizeof(BlockHeader);
            prev->next = current->next;
            current = prev;
            if (allocator->free_list == current) allocator->free_list = prev;
            continue;
        }
        prev = current;
        current = current->next;
    }
```

```
}

void allocator_destroy(Allocator *allocator) {
    if (allocator) {
        munmap(allocator->base_addr, allocator->total_size + sizeof(Allocator));
    }
}
```

Протокол работы программы:

```
aleksandrchuvilov@MacBook-Pro-Aleksandr src % ./main ./list.so
Allocated block at 830c59201
Freed block
aleksandrchuvilov@MacBook-Pro-Aleksandr src % ./main ./buddy.so
Allocated block at 0214a3201
Freed block
aleksandrchuvilov@MacBook-Pro-Aleksandr src % ./main ./list.so
Allocated block at 83085d401
Freed block
aleksandrchuvilov@MacBook-Pro-Aleksandr src % ./main ./list.so
Allocated block at 830c8b001
Freed block
```

Strace:

strace -f ./main ./list.so
execve("./main", ["./main", "./list.so"], 0x7ffd98519b40 /* 46 vars */) = 0
brk(NULL) = 0x55d578a8a000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffffd993f00) = -1 EINVAL (Недопустимый аргумент)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x700c29481000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (Нет такого файла или каталога)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=58047, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 58047, PROT_READ, MAP_PRIVATE, 3, 0) = 0x700c29472000
close(3) = 0
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\0P\237\2\0\0\0\0\0"..., 832) = 832
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 \0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0\0"..., 48, 848) = 48
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0I\17\357\204\3$\f\221\2039x\324\224\323\236S"..., 68, 896) = 68
newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=2220400, ...}, AT_EMPTY_PATH) = 0
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
mmap(NULL, 2264656, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x700c29200000
mprotect(0x700c29228000, 2023424, PROT_NONE) = 0
mmap(0x700c29228000, 1658880, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x700c29228000
mmap(0x700c293bd000, 360448, PROT_READ,

```
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1bd000) = 0x700c293bd000
mmap(0x700c29416000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x215000) = 0x700c29416000
mmap(0x700c2941c000, 52816, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x700c2941c000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x700c2946f000
arch_prctl(ARCH_SET_FS, 0x700c2946f740) = 0
set_tid_address(0x700c2946fa10)         = 6395
set_robust_list(0x700c2946fa20, 24)     = 0
rseq(0x700c294700e0, 0x20, 0, 0x53053053) = 0
mprotect(0x700c29416000, 16384, PROT_READ) = 0
mprotect(0x55d57754f000, 4096, PROT_READ) = 0
mprotect(0x700c294bb000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024,
rlim_max=RLIM64_INFINITY}) = 0
munmap(0x700c29472000, 58047)           = 0
getrandom("\x68\xb8\xb0\x21\x82\xc7\x7c\xfd", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x55d578a8a000
brk(0x55d578aab000)                     = 0x55d578aab000
openat(AT_FDCWD, "./list.so", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0"..., 832) = 832
newfstatat(3, "", {st_mode=S_IFREG|0775, st_size=15640, ...}, AT_EMPTY_PATH) =
0
getcwd("/home/artemdelgray/\320\227\320\260\320\263\321\200\321\203\320\267\320\2
72\320\270/Telegram Desktop", 128) = 53
mmap(NULL, 16432, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x700c2947c000
mmap(0x700c2947d000, 4096, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1000) = 0x700c2947d000
mmap(0x700c2947e000, 4096, PROT_READ,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x700c2947e000
mmap(0x700c2947f000, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x700c2947f000
close(3)                                = 0
mprotect(0x700c2947f000, 4096, PROT_READ) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x700c294ba000
write(1, "Memory allocated successfully\n", 30Memory allocated successfully
) = 30
write(1, "Memory freed\n", 13Memory freed
)       = 13
munmap(0x700c294ba000, 4096)            = 0
write(1, "Program exited successfully\n", 28Program exited successfully
) = 28
exit_group(0)                           = ?
+++ exited with 0 +++
```

Вывод

В ходе лабораторной работы была разработана программа, которая демонстрирует работу аллокатора. Было реализовано 2. Данные программы предоставляют доступ к управлению памятью, демонстрируют разные алгоритмы, направленные на решение одной из ключевых задач в программировании — эффективного использования ограниченных ресурсов памяти. Каждый из них имеет свои особенности, плюсы и минусы, что позволяет выбрать наиболее подходящий метод в зависимости от конкретных условий.