

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

**Курсовая работа по курсу**

**«Операционные системы»**

Группа: М8О-209БВ-24

Студент: Касеева Я.М.

Преподаватель: Миронов Е.С.

Оценка: \_\_\_\_\_

Дата: 16.12.25

Москва, 2025

# Постановка задачи

## Вариант 15.

Исследование 2 аллокаторов памяти (списки свободных блоков (первое подходящее) и алгоритм двойников): необходимо реализовать два алгоритма аллокации памяти и сравнить их по следующим характеристикам:

- Фактор использования
- Скорость выделения блоков
- Скорость освобождения блоков
- Простота использования аллокатора

Каждый аллокатор памяти должен иметь функции аналогичные стандартным функциям `free` и `malloc` (`realloc`, опционально) или он должен быть реализован в рамках концепции `std::allocator`. Данный выбор производится преподавателем группы. Перед работой каждый аллокатор инициализируется свободными страницами памяти, выделенными стандартными средствами ядра. Необходимо самостоятельно разработать стратегию тестирования для определения ключевых характеристик аллокаторов памяти. При тестировании нужно свести к минимуму потери точности из-за накладных расходов при измерении ключевых характеристик, описанных выше.

## Общий метод и алгоритм решения

### Использованные системные вызовы:

- `execve()` — запуск исполняемого файла программы
- `mmap()` — выделение и отображение памяти в адресное пространство процесса
- `munmap()` — освобождение ранее выделенных областей памяти
- `brk()` — управление кучей процесса
- `mprotect()` — изменение прав доступа к участкам памяти
- `openat()` — открытие файлов и библиотек
- `read()` — чтение данных из файлов
- `close()` — закрытие файловых дескрипторов
- `clock_gettime()` — измерение времени работы аллокаторов
- `write()` — вывод результатов тестирования
- `exit_group()` — завершение работы программы

## Алгоритм работы программы:

Программа предназначена для сравнения двух алгоритмов аллокации памяти: аллокатора на основе списков свободных блоков (First Fit) и аллокатора, реализованного по алгоритму двойников (Buddy System).

### Описание исследуемых алгоритмов:

Аллокатор на основе списков свободных блоков хранит информацию о свободных участках памяти в виде связного списка. При запросе памяти осуществляется последовательный поиск первого блока, размер которого достаточен для удовлетворения запроса. В случае успешного выделения блок уменьшается или разбивается, а при освобождении памяти блок возвращается в список свободных. Данный подход прост в реализации, но может приводить к фрагментации памяти.

Аллокатор типа Buddy System использует память, размер которой является степенью двойки. При запросе памяти выбирается минимальный блок подходящего размера, при необходимости выполняется рекурсивное деление блоков на пары (двойники). При освобождении памяти производится попытка объединения освобождённого блока с его двойником, что позволяет уменьшить фрагментацию и упростить управление памятью.

### Процесс тестирования:

Для каждого аллокатора предварительно выделяется фиксированный пул памяти. Далее выполняется серия операций выделения памяти случайного размера в диапазоне от 16 до 271 байта. После завершения всех операций выделения производится освобождение всех ранее выделенных блоков.

В процессе тестирования измеряется:

- время выполнения операций выделения памяти;
- время выполнения операций освобождения памяти;
- объём используемой и свободной памяти после этапа выделения.

### Обоснование подхода тестирования:

Использование случайных размеров блоков позволяет смоделировать поведение аллокаторов в условиях, приближённых к реальным сценариям работы программ. Фиксированный объём памяти и одинаковая последовательность операций обеспечивают корректность сравнения алгоритмов. Измерение времени производится с использованием системного вызова `clock_gettime`, что позволяет получить достаточно точные результаты с минимальными накладными расходами.

## Код программы

### allocator.h

```
#ifndef ALLOCATOR_H
#define ALLOCATOR_H

#include <stddef.h>

typedef struct Allocator Allocator;

#endif
```

## freelist.h

```
#ifndef FREELIST_H
#define FREELIST_H

#include "allocator.h"

Allocator* freelist_create(void *memory, size_t size);

void* freelist_alloc(Allocator *a, size_t size);
void freelist_free(Allocator *a, void *ptr);

size_t freelist_get_used_memory(Allocator *a);
size_t freelist_get_free_memory(Allocator *a);

#endif
```

## freelist.c

```
#include "freelist.h"
#include <stddef.h>
#include <stdint.h>

typedef struct Block {
    size_t size;
    struct Block *next;
} Block;

struct Allocator {
    Block *free_list;
    size_t total_memory;
    size_t used_memory;
};

Allocator* freelist_create(void *memory, size_t size) {
    Allocator *a = (Allocator*)memory;
    a->total_memory = size - sizeof(Allocator);
    a->used_memory = 0;

    a->free_list = (Block*)((char*)memory + sizeof(Allocator));
    a->free_list->size = a->total_memory;
    a->free_list->next = NULL;
    return a;
}

void* freelist_alloc(Allocator *a, size_t size) {
    Block *prev = NULL;
    Block *cur = a->free_list;

    while (cur) {
        if (cur->size >= size + sizeof(Block)) {
            Block *next = (Block*)((char*)cur + sizeof(Block) + size);
            next->size = cur->size - size - sizeof(Block);
            next->next = cur->next;

            if (prev) prev->next = next;
            else a->free_list = next;

            cur->size = size;
            a->used_memory += size + sizeof(Block);
            return (char*)cur + sizeof(Block);
        }
        prev = cur;
        cur = cur->next;
    }
    return NULL;
}
```

```

}

void freelist_free(Allocator *a, void *ptr) {
    if (!ptr) return;
    Block *b = (Block*)((char*)ptr - sizeof(Block));
    a->used_memory -= b->size + sizeof(Block);
    b->next = a->free_list;
    a->free_list = b;
}

size_t freelist_get_used_memory(Allocator *a) {
    return a->used_memory;
}

size_t freelist_get_free_memory(Allocator *a) {
    return a->total_memory - a->used_memory;
}

```

## buddy.h

```

#ifndef BUDDY_H
#define BUDDY_H

#include "allocator.h"

Allocator* buddy_create(void *memory, size_t size);

void* buddy_alloc(Allocator *a, size_t size);
void buddy_free(Allocator *a, void *ptr);

size_t buddy_get_used_memory(Allocator *a);
size_t buddy_get_free_memory(Allocator *a);

#endif

```

## buddy.c

```

#include "buddy.h"
#include <stddef.h>
#include <stdint.h>
#include <string.h>

#define MIN_ORDER 4
#define MAX_ORDER 20

typedef struct BuddyBlock {
    struct BuddyBlock *next;
    size_t order;
} BuddyBlock;

struct Allocator {
    BuddyBlock *free_lists[MAX_ORDER + 1];
    void *memory_start;
    size_t total_memory;
    size_t used_memory;
};

static size_t size_to_order(size_t size) {
    size_t order = MIN_ORDER;
    size_t block_size = 1UL << order;
    while (block_size < size + sizeof(BuddyBlock)) {
        block_size <= 1;
        order++;
    }
}

```

```

    return order;
}

static uintptr_t get_offset(void *base, void *ptr) {
    return (uintptr_t)ptr - (uintptr_t)base;
}

static BuddyBlock* get_buddy(void *base, BuddyBlock *block) {
    uintptr_t offset = get_offset(base, block);
    uintptr_t buddy_offset = offset ^ (1UL << block->order);
    return (BuddyBlock*)((char*)base + buddy_offset);
}

Allocator* buddy_create(void *memory, size_t size) {
    Allocator *a = (Allocator*)memory;
    memset(a->free_lists, 0, sizeof(a->free_lists));
    a->memory_start = memory;
    a->total_memory = size - sizeof(Allocator);
    a->used_memory = 0;

    size_t order = size_to_order(a->total_memory);
    BuddyBlock *b = (BuddyBlock*)((char*)memory + sizeof(Allocator));
    b->next = NULL;
    b->order = order;
    a->free_lists[order] = b;

    return a;
}

void* buddy_alloc(Allocator *a, size_t size) {
    size_t order = size_to_order(size);
    size_t i = order;

    while (i <= MAX_ORDER && a->free_lists[i] == NULL) i++;
    if (i > MAX_ORDER) return NULL;

    BuddyBlock *block = a->free_lists[i];
    a->free_lists[i] = block->next;

    while (i > order) {
        i--;
        BuddyBlock *buddy = (BuddyBlock*)((char*)block + (1UL << i));
        buddy->next = a->free_lists[i];
        buddy->order = i;
        a->free_lists[i] = buddy;
    }

    block->order = order;
    block->next = NULL;
    a->used_memory += (1UL << order);

    return (char*)block + sizeof(BuddyBlock);
}

void buddy_free(Allocator *a, void *ptr) {
    if (!ptr) return;

    BuddyBlock *block = (BuddyBlock*)((char*)ptr - sizeof(BuddyBlock));
    size_t order = block->order;

    a->used_memory -= (1UL << order);

    while (order < MAX_ORDER) {
        BuddyBlock **list = &a->free_lists[order];
        BuddyBlock *prev = NULL;
        BuddyBlock *cur = *list;
        BuddyBlock *buddy = get_buddy(a->memory_start, block);
    }
}

```

```

    int merged = 0;

    while (cur) {
        if (cur == buddy) {
            if (prev) prev->next = cur->next;
            else *list = cur->next;

            if ((uintptr_t)block > (uintptr_t)cur) block = cur;

            order++;
            block->order = order;
            merged = 1;
            break;
        }
        prev = cur;
        cur = cur->next;
    }
    if (!merged) break;
}

block->next = a->free_lists[block->order];
a->free_lists[block->order] = block;
}

size_t buddy_get_used_memory(Allocator *a) {
    return a->used_memory;
}

size_t buddy_get_free_memory(Allocator *a) {
    return a->total_memory - a->used_memory;
}

```

## main.c

```

#include "freelist.h"
#include "buddy.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MEMORY_SIZE (1024*1024)
#define N 100000

typedef struct {
    double alloc_time;
    double free_time;
    size_t used_mem;
    size_t free_mem;
} Stats;

static Stats test_allocator(const char *name,
                           Allocator *A,
                           void* (*alloc_func)(Allocator*, size_t),
                           void (*free_func)(Allocator*, void*),
                           size_t (*get_used)(Allocator*),
                           size_t (*get_free)(Allocator*)) {

    (void)name;

    void *ptrs[N];
    size_t sizes[N];
    for (int i = 0; i < N; i++) sizes[i] = (rand() % 256) + 16;

    Stats s = {0};

    clock_t start = clock();
    for (int i = 0; i < N; i++)

```

```

        ptrs[i] = alloc_func(A, sizes[i]);
        clock_t end = clock();
        s.alloc_time = (double)(end - start) / CLOCKS_PER_SEC;

        s.used_mem = get_used(A);
        s.free_mem = get_free(A);

        start = clock();
        for (int i = 0; i < N; i++)
            free_func(A, ptrs[i]);
        end = clock();
        s.free_time = (double)(end - start) / CLOCKS_PER_SEC;

        return s;
}

int main(void) {
    srand((unsigned)time(NULL));

    void *mem_ff = malloc(MEMORY_SIZE);
    void *mem_bd = malloc(MEMORY_SIZE);

    Allocator *ff = freelist_create(mem_ff, MEMORY_SIZE);
    Allocator *bd = buddy_create(mem_bd, MEMORY_SIZE);

    Stats s_ff = test_allocator("Free List", ff, freelist_alloc, freelist_free,
                                freelist_get_used_memory, freelist_get_free_memory);
    Stats s_bd = test_allocator("Buddy", bd, buddy_alloc, buddy_free,
                                buddy_get_used_memory, buddy_get_free_memory);

    printf("=== Сравнение аллокаторов ===\n");
    printf("%-10s | %-11s | %-10s | %-10s\n",
            "Allocator", "Utilization", "Alloc Time", "Free Time");
    printf("-----\n");

    printf("%-10s | %-10.2f%% | %-10.6f | %-10.6f\n",
            "Free List",
            100.0 * s_ff.used_mem / (s_ff.used_mem + s_ff.free_mem),
            s_ff.alloc_time,
            s_ff.free_time);

    printf("%-10s | %-10.2f%% | %-10.6f | %-10.6f\n",
            "Buddy",
            100.0 * s_bd.used_mem / (s_bd.used_mem + s_bd.free_mem),
            s_bd.alloc_time,
            s_bd.free_time);

    free(mem_ff);
    free(mem_bd);

    return 0;
}

```

## Makefile

```

CC = gcc
CFLAGS = -Wall -Wextra -O2 -std=c99

TARGET = run
OBJ = main.o freelist.o buddy.o

all: $(TARGET)

$(TARGET): $(OBJ)
    $(CC) $(CFLAGS) $(OBJ) -o $(TARGET)

```



```

main.o: main.c freelist.h buddy.h allocator.h
$ (CC) $(CFLAGS) -c main.c

freelist.o: freelist.c freelist.h allocator.h
$ (CC) $(CFLAGS) -c freelist.c

buddy.o: buddy.c buddy.h allocator.h
$ (CC) $(CFLAGS) -c buddy.c

run_program: $(TARGET)
./$(TARGET)

clean:
rm -f *.o $(TARGET)

```

## Протокол работы программы

### Тестирование 1:

(base) yanakasaeva@MacBook-Air--YanaK src % ./run

=== Сравнение аллокаторов ===

Allocator	Utilization	Alloc Time	Free Time
Free List	100.00 %	0.001602	0.000301
Buddy	100.02 %	0.002512	0.011661

### Strace:

```
1117 **execve**("./run", ["/run"], 0xffffdae4f3c8 /* 8 vars */) = 0
```

```
1117 **brk**(NULL) = 0xaaaaeaa40000
```

```
1117 mmap(NULL, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xffffb61ea000
```

```
1117 faccessat(AT_FDCWD, "/etc/ld.so.preload", R_OK) = -1 ENOENT
```

```
1117 openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
```

```
1117 fstat(3, {st_mode=S_IFREG|0644, st_size=8467, ...}) = 0
```

```
1117 mmap(NULL, 8467, PROT_READ, MAP_PRIVATE, 3, 0) = 0xffffb61e7000
```

```
1117 close(3) = 0
```

```
1117 openat(AT_FDCWD, "/lib/aarch64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
```

```
1117 read(3, "\177ELF...", 832) = 832
```

```
1117 fstat(3, {st_mode=S_IFREG|0755, st_size=1722920, ...}) = 0
```

```
1117 mmap(NULL, 1892240, PROT_NONE,
MAP_PRIVATE|MAP_ANONYMOUS|MAP_DENYWRITE, -1, 0) = ...
```

```

1117 mmap(0xffffb5ff0000, 1826704, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0) = ...
1117 munmap(0xffffb5fe3000, 53248) = 0
1117 munmap(0xffffb61ae000, 12176) = 0
1117 **brk**(0xaaaaeaa61000)          = 0xaaaaeaa61000
1117 **mmap**(NULL, 1052672, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xffffb5eef000
1117 **mmap**(NULL, 1052672, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xffffb5dee000
1117 **clock_gettime**(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=965957}) = 0
1117 **clock_gettime**(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=9843581}) = 0

1117 **write**(1, "=== Сравнение аллокаторов ===\n", 50) = 50
1117 **write**(1, "Free List | 100.00  % | ...", 51) = 51
1117 **write**(1, "Buddy   | 100.02  % | ...", 51) = 51
1117 **munmap**(0xffffb5eef000, 1052672) = 0
1117 **munmap**(0xffffb5dee000, 1052672) = 0
1117 **exit_group** (0)

```

## Вывод

В ходе выполнения курсового проекта были спроектированы и реализованы два программных прототипа аллокаторов памяти: аллокатор на основе списков свободных блоков (First Fit) и аллокатор, реализованный по алгоритму двойников (Buddy System). Для каждого аллокатора был разработан единый интерфейс и проведено тестирование на одинаковом наборе операций выделения и освобождения памяти.

В результате тестирования было выполнено сравнение аллокаторов по фактору использования памяти, скорости выделения и освобождения блоков, а также по особенностям их практического применения. Полученные результаты показали, что аллокатор на основе списков свободных блоков отличается простотой реализации и высокой скоростью освобождения памяти, однако подвержен фрагментации. Аллокатор Buddy System обеспечивает более структурированное управление памятью и эффективное объединение свободных блоков, но требует больших накладных расходов на операции выделения и освобождения.

Таким образом, выбор алгоритма аллокации памяти зависит от требований к производительности и характеру нагрузки. Проведённое исследование позволило на практике изучить особенности работы различных алгоритмов управления памятью и углубить понимание механизмов, используемых в операционных системах.