

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: Касаева 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)) {
        if (order >= MAX_ORDER) return 0;
        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) {
    if (size < (1UL << MIN_ORDER) + sizeof(Allocator)) return NULL;

    Allocator *a = (Allocator*)memory;
    memset(a->free_lists, 0, sizeof(a->free_lists));

    // ИСПРАВЛЕНИЕ 1: memory_start указывает на начало пользовательской памяти
    a->memory_start = (char*)memory + sizeof(Allocator);

    // ИСПРАВЛЕНИЕ 2: total_memory - только пользовательская память
    a->total_memory = size - sizeof(Allocator);

    // ИСПРАВЛЕНИЕ 3: used_memory начинается с 0
    a->used_memory = 0;

    size_t max_possible = a->total_memory;
    size_t order = MIN_ORDER;
    while (order < MAX_ORDER && (1UL << (order + 1)) <= max_possible) {
        order++;
    }

    BuddyBlock *b = (BuddyBlock*)a->memory_start;
    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);
    if (order == 0) return NULL;

    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 || !a) return;

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

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

    while (order < MAX_ORDER) {
        BuddyBlock *buddy = get_buddy(a->memory_start, block);
        BuddyBlock **list = &a->free_lists[order];
        BuddyBlock *cur = *list;
        BuddyBlock *prev = NULL;
        int merged = 0;

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

                if ((uintptr_t)block > (uintptr_t)buddy) {
                    block = buddy;
                }

                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) {
    size_t free_mem = 0;
    for (int i = MIN_ORDER; i <= MAX_ORDER; i++) {
        BuddyBlock *block = a->free_lists[i];
        while (block) {
            free_mem += (1UL << i);
            block = block->next;
        }
    }
    return free_mem;
}

```

## main.c

```

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

```

```

#include <unistd.h>

#define MEMORY_SIZE (1024*1024)
#define N 10000

typedef struct {
    double alloc_time;
    double free_time;
    size_t used_mem;
    size_t free_mem;
    int successful_allocs;
} 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 *ptrs[N] = {0};
    size_t sizes[N];
    int successful_allocs = 0;

    for (int i = 0; i < N; i++) {
        sizes[i] = (rand() % 112) + 16;
    }

    Stats s = {0};

    clock_t start = clock();
    for (int i = 0; i < N; i++) {
        ptrs[i] = alloc_func(A, sizes[i]);
        if (ptrs[i]) successful_allocs++;
    }
    clock_t end = clock();
    s.alloc_time = (double)(end - start) / CLOCKS_PER_SEC;
    s.successful_allocs = successful_allocs;

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

    start = clock();
    for (int i = 0; i < N; i++) {
        if (ptrs[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));

    size_t page_size = sysconf(_SC_PAGESIZE);
    size_t aligned_size = ((MEMORY_SIZE + page_size - 1) / page_size) * page_size;

    void *mem_ff = mmap(NULL, aligned_size,
                        PROT_READ | PROT_WRITE,
                        MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    void *mem_bd = mmap(NULL, aligned_size,
                        PROT_READ | PROT_WRITE,
                        MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

    if (mem_ff == MAP_FAILED || mem_bd == MAP_FAILED) {

```



```

    perror("mmap failed");
    return 1;
}

Allocator *ff = freelist_create(mem_ff, aligned_size);
Allocator *bd = buddy_create(mem_bd, aligned_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("%-10s | %-11s | %-12s | %-12s\n",
        "Allocator", "Utilization", "Alloc Time", "Free Time");
printf("-----\n");

printf("%-10s | %-10.2f%% | %-12.6f | %-12.6f\n",
        "Free List",
        100.0 * s_ff.used_mem / aligned_size,
        s_ff.alloc_time,
        s_ff.free_time);

printf("%-10s | %-10.2f%% | %-12.6f | %-12.6f\n",
        "Buddy",
        100.0 * s_bd.used_mem / aligned_size,
        s_bd.alloc_time,
        s_bd.free_time);

munmap(mem_ff, aligned_size);
munmap(mem_bd, aligned_size);

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

Allocator | Utilization | Alloc Time | Free Time

-----  
Free List | 83.15 % | 0.000162 | 0.000063

Buddy | 50.00 % | 0.000289 | 0.000131

## 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

Реализация аллокаторов памяти является важной задачей в системном программировании. В данной работе были разработаны два алгоритма управления памятью: Buddy System и Free List. Оба аллокатора используют технику memory mapping через системный вызов mmap для выделения начального пула памяти. Целью исследования было сравнение эффективности этих алгоритмов на случайной рабочей нагрузке.

Методика тестирования включала выполнение 10000 операций аллокации и освобождения блоков памяти случайного размера. Эксперимент проводился на выделенном регионе памяти размером 1 МБ. Для обеспечения воспроизводимости результатов использовался детерминированный генератор случайных чисел.

Free List аллокатор продемонстрировал превосходную эффективность, успешно выполнив все 10000 запросов на выделение памяти. Утилизация памяти составила 83.15%, что свидетельствует о незначительных накладных расходах. Время выполнения операций было минимальным: 0.000162 секунд на аллокацию и 0.000063 секунд на освобождение.

Buddy System показал более скромные результаты из-за присущей данному алгоритму внутренней фрагментации. Утилизация памяти составила всего 50.00%, что объясняется требованием выравнивания блоков по степеням двойки. Хотя алгоритм успешно обработал запросы, его эффективность на случайных мелких аллокациях оказалась ниже.

Результаты подтверждают теоретические ожидания: Buddy System лучше подходит для систем с предсказуемым размером блоков, тогда как Free List более адаптивен к случайной рабочей нагрузке. Оба алгоритма корректно интегрированы с механизмом memory mapping, что позволяет использовать их в реальных системных приложениях.