

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

Студентка: Полевая А.О.

Преподаватель: Миронов Е.С.

Оценка: _____

Дата: 18.12.25

Москва 2025

Постановка задачи

Вариант 21

Исследование 2 аллокаторов памяти: 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, дополнительно) 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

Необходимо сравнить два алгоритма аллокации: **списки свободных блоков (наиболее подходящее)** и **алгоритм двойников**.

Общий метод и алгоритм решения

Использованные системные вызовы:

- mmap () — выделение виртуальной памяти у ОС
- munmap () — освобождение памяти обратно ОС

Алгоритм работы программы

Программа реализует и сравнивает работу двух пользовательских аллокаторов памяти — аллокатора со списком свободных блоков (Free List) и аллокатора двойников (Buddy Allocator). При инициализации каждому аллокатору передаётся заранее выделенный непрерывный участок памяти, внутри которого он самостоятельно управляет размещением и освобождением блоков.

Аллокатор со списком свободных блоков (Free List, стратегия best-fit).

Память представляется в виде связного списка свободных блоков, заголовок каждого блока хранится непосредственно внутри управляемой области памяти и содержит его размер и указатель на следующий элемент списка. При запросе на выделение памяти аллокатор последовательно просматривает весь список свободных блоков и выбирает блок минимального размера, который способен удовлетворить запрос (стратегия best-fit). Если размер найденного блока существенно больше требуемого, он разделяется на занятый блок и новый свободный блок, который остаётся в списке. При освобождении памяти блок возвращается в список свободных, который поддерживается упорядоченным по адресам, и при возможности выполняется слияние соседних свободных блоков для уменьшения внешней фрагментации.

Аллокатор двойников (Buddy Allocator).

В buddy-аллокаторе управляемая память разбивается на блоки размеров, равных степеням двойки, и для каждого размера поддерживается отдельный список свободных блоков. При выделении памяти определяется минимальный порядок блока, способный вместить запрошенный размер, после чего при необходимости больший свободный блок рекурсивно делится на два равных «двойника» до достижения нужного размера. Один из полученных блоков возвращается пользователю, а остальные помещаются в соответствующие списки свободных блоков. При освобождении памяти аллокатор пытается найти свободного «двойника» освобождаемого блока; если он найден, оба блока объединяются в один большего порядка, и процесс повторяется рекурсивно. Такая стратегия обеспечивает быстрое выделение и освобождение памяти и эффективное объединение блоков, но может приводить к внутренней фрагментации из-за округления размеров к степеням двойки.

Процесс тестирования

Процесс тестирования в программе организован в основной функции и направлен на сравнение производительности и эффективности двух реализованных аллокаторов памяти. Сначала выделяются два независимых непрерывных участка памяти фиксированного размера, по одному для каждого аллокатора, после чего на их основе создаются Free List и Buddy аллокаторы. Далее для каждого аллокатора выполняется серия из большого числа однотипных операций: в цикле производится последовательное выделение блоков одинакового размера до тех пор, пока память не закончится или не будет достигнуто заданное число итераций, при этом фиксируется время выполнения операций выделения и подсчитывается фактически используемая память. После этого все ранее выделенные блоки освобождаются в отдельном цикле, и также измеряется время выполнения операций освобождения памяти.

Обоснование подхода тестирования

Выбранный подход тестирования позволяет получить наглядное и корректное сравнение различных стратегий управления памятью в одинаковых условиях. Оба аллокатора работают с одинаковым объёмом заранее выделенной памяти и подвергаются одинаковой нагрузке, что исключает влияние внешних факторов и позволяет сравнивать только особенности самих алгоритмов. Раздельное измерение времени выделения и освобождения памяти позволяет оценить эффективность каждой операции в отдельности, а расчёт процента используемой памяти даёт представление о степени фрагментации и рациональности использования доступного объёма.

Код программы

main.h

```
#ifndef ALLOCATOR_H
#define ALLOCATOR_H

#include <stddef.h>

typedef enum {
```

```

    ALLOC_FREE_LIST,
    ALLOC_BUDDY
} AllocatorType;

typedef struct Allocator Allocator;

struct Allocator {
    AllocatorType type;
    void *impl;
    size_t total_size;
    size_t used_size;
};

Allocator* createMemoryAllocator(AllocatorType type, size_t
memory_size);

void* allocator_alloc(Allocator *allocator, size_t size);
void allocator_free(Allocator *allocator, void *ptr);
void allocator_destroy(Allocator *allocator);

#endif

```

main.c

```

#include "main.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <stdint.h>
#include <sys/mman.h>
#include <unistd.h>

//Аллокатор со списком свободных блоков
typedef struct FreeBlock {
    size_t size;
    struct FreeBlock *next;
} FreeBlock;

typedef struct {
    void *memory;
    size_t size;
    FreeBlock *free_list;
} FreeListAllocator;

static FreeListAllocator* freelist_create(void *memory, size_t size) {
    FreeListAllocator *alloc = malloc(sizeof(FreeListAllocator));
    alloc->memory = memory;
    alloc->size = size;
}

```

```

    alloc->free_list = (FreeBlock*)memory;
    alloc->free_list->size = size - sizeof(FreeBlock);
    alloc->free_list->next = NULL;
    return alloc;
}

static void* freelist_alloc(Allocator *base, size_t size) {
    FreeListAllocator *alloc = base->impl;
    FreeBlock *best = NULL;
    FreeBlock *best_prev = NULL;
    FreeBlock *prev = NULL;
    FreeBlock *curr = alloc->free_list;

    while (curr) {
        if (curr->size >= size) {
            if (!best || curr->size < best->size) {
                best = curr;
                best_prev = prev;
            }
        }
        prev = curr;
        curr = curr->next;
    }

    if (!best) {
        return NULL;
    }

    if (best->size <= size + sizeof(FreeBlock)) {
        if (best_prev) {
            best_prev->next = best->next;
        } else {
            alloc->free_list = best->next;
        }
        base->used_size += best->size;
        return (void*)(best + 1);
    }

    FreeBlock *next = (FreeBlock*)((char*)(best + 1) + size);
    next->size = best->size - size - sizeof(FreeBlock);
    next->next = best->next;

    if (best_prev) {
        best_prev->next = next;
    } else {
        alloc->free_list = next;
    }

    best->size = size;
    base->used_size += size;
    return (void*)(best + 1);
}

```

```

}

static void freelist_free(Allocator *base, void *ptr) {
    if (!ptr) {
        return;
    }
    FreeListAllocator *alloc = base->impl;
    FreeBlock *block = ((FreeBlock*)ptr) - 1;
    base->used_size -= block->size;

    // вставка в список с упорядочиванием по адресу
    FreeBlock **curr = &alloc->free_list;
    while (*curr && *curr < block) {
        curr = &(*curr)->next;
    }
    block->next = *curr;
    *curr = block;

    // слияние с соседями
    if (block->next && (char*)(block + 1) + block->size == (char*)block->next) {
        block->size += sizeof(FreeBlock) + block->next->size;
        block->next = block->next->next;
    }
    if (curr != &alloc->free_list) {
        FreeBlock *prev = alloc->free_list;
        while (prev && prev->next != block) {
            prev = prev->next;
        }

        if (prev && (char*)(prev + 1) + prev->size == (char*)block) {
            prev->size += sizeof(FreeBlock) + block->size;
            prev->next = block->next;
        }
    }
}

//Аллокатор двойников
#define MAX_ORDER 16

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

typedef struct {
    void *memory;
    int max_order;
    BuddyBlock *free_lists[MAX_ORDER + 1];
} BuddyAllocator;

```

```

static int order_for_size(size_t size) {
    int order = 0;
    size_t block = 1;
    while (block < size) {
        block <<= 1;
        order++;
    }
    return order;
}

static BuddyAllocator* buddy_create(void *memory, size_t size) {
    BuddyAllocator *alloc = malloc(sizeof(BuddyAllocator));
    alloc->memory = memory;
    alloc->max_order = order_for_size(size);
    for (int i = 0; i <= MAX_ORDER; i++) {
        alloc->free_lists[i] = NULL;
    }
    alloc->free_lists[alloc->max_order] = (BuddyBlock*)memory;
    alloc->free_lists[alloc->max_order]->next = NULL;
    return alloc;
}

static void* buddy_alloc(Allocator *base, size_t size) {
    BuddyAllocator *alloc = base->impl;
    int order = order_for_size(size + sizeof(int));
    int current = order;

    while (current <= alloc->max_order && alloc->free_lists[current] == NULL) {
        current++;
    }

    if (current > alloc->max_order) {
        return NULL;
    }

    BuddyBlock *block = alloc->free_lists[current];
    alloc->free_lists[current] = block->next;

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

    int *header = (int*)block;
    *header = order;
    base->used_size += (1 << order);
    return (void*)(header + 1);
}

```

```

static void buddy_free(Allocator *base, void *ptr) {
    if (!ptr) {
        return;
    }
    BuddyAllocator *alloc = base->impl;
    int *header = ((int*)ptr) - 1;
    int order = *header;
    BuddyBlock *block = (BuddyBlock*)header;
    base->used_size -= (1 << order);

    while (order < alloc->max_order) {
        uintptr_t offset = (uintptr_t)block - (uintptr_t)alloc->memory;
        uintptr_t buddy_offset = offset ^ (1 << order);
        BuddyBlock *buddy = (BuddyBlock*)((uintptr_t)alloc->memory +
buddy_offset);

        BuddyBlock **curr = &alloc->free_lists[order];
        while (*curr && *curr != buddy) {
            curr = &(*curr)->next;
        }
        if (*curr != buddy) {
            break;
        }

        *curr = buddy->next;
        if (buddy < block) {
            block = buddy;
        }
        order++;
    }
    block->next = alloc->free_lists[order];
    alloc->free_lists[order] = block;
}

static size_t power_of_two(size_t size) {
    size_t power = 1;
    while (power < size) {
        power *= 2;
    }
    return power;
}

//AllocatorCreate
Allocator* createMemoryAllocator(AllocatorType type, size_t memory_size) {
    Allocator *allocator = malloc(sizeof(Allocator));
    if (!allocator) return NULL;

    size_t real_size = memory_size;
}

```

```

    if (type == ALLOC_BUDDY) {
        real_size = power_of_two(memory_size);
    }

    void *memory = mmap(NULL, real_size, PROT_READ | PROT_WRITE,
MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (memory == MAP_FAILED) {
        free(allocator);
        return NULL;
    }

    allocator->type = type;
    allocator->total_size = real_size;
    allocator->used_size = 0;

    if (type == ALLOC_FREE_LIST) {
        allocator->impl = freelist_create(memory, real_size);
    } else {
        allocator->impl = buddy_create(memory, real_size);
    }

    return allocator;
}

//Единые функции
void* allocator_alloc(Allocator *allocator, size_t size) {
    if (allocator->type == ALLOC_FREE_LIST) {
        return freelist_alloc(allocator, size);
    } else {
        return buddy_alloc(allocator, size);
    }
}

void allocator_free(Allocator *allocator, void *ptr) {
    if (allocator->type == ALLOC_FREE_LIST) {
        freelist_free(allocator, ptr);
    } else {
        buddy_free(allocator, ptr);
    }
}

void allocator_destroy(Allocator *allocator) {
    if (!allocator) {
        return;
    }

    void *memory;
    if (allocator->type == ALLOC_FREE_LIST) {
        FreeListAllocator *alloc = allocator->impl;

```

```

        memory = alloc->memory;
        free(alloc);
    } else {
        BuddyAllocator *alloc = allocator->impl;
        memory = alloc->memory;
        free(alloc);
    }

    munmap(memory, allocator->total_size);
    free(allocator);
}

//Тестирование
#define MEMORY_SIZE 65536
#define ITERATIONS 1000
#define BLOCK_SIZE 32

int main() {
    Allocator *fl = createMemoryAllocator(ALLOC_FREE_LIST, MEMORY_SIZE);
    Allocator *bd = createMemoryAllocator(ALLOC_BUDDY, MEMORY_SIZE);

    void *ptrs[ITERATIONS];
    int count;
    clock_t start, end;

    //Free List alloc
    count = 0;
    start = clock();
    for (int i = 0; i < ITERATIONS; i++) {
        ptrs[i] = allocator_alloc(fl, BLOCK_SIZE);
        if (!ptrs[i]) {
            break;
        }
        count++;
    }
    end = clock();

    printf("FreeList alloc time: %lf\n", (double)(end - start) /
CLOCKS_PER_SEC);
    printf("FreeList usage: %.2f%%\n", 100.0 * fl->used_size / fl-
>total_size);

    //Free List free
    start = clock();
    for (int i = 0; i < count; i++) {
        allocator_free(fl, ptrs[i]);
    }
    end = clock();
}

```

```

    printf("FreeList free time: %lf\n", (double)(end - start) /
CLOCKS_PER_SEC);

    //Buddy alloc
    count = 0;
    start = clock();
    for (int i = 0; i < ITERS; i++) {
        ptrs[i] = allocator_alloc(bd, BLOCK_SIZE);
        if (!ptrs[i]) {
            break;
        }
        count++;
    }
    end = clock();

    printf("Buddy alloc time: %lf\n", (double)(end - start) /
CLOCKS_PER_SEC);
    printf("Buddy usage: %.2f%%\n", 100.0 * bd->used_size / bd-
>total_size);

    //Buddy free
    start = clock();
    for (int i = 0; i < count; i++) {
        allocator_free(bd, ptrs[i]);
    }
    end = clock();

    printf("Buddy free time: %lf\n", (double)(end - start) /
CLOCKS_PER_SEC);

    allocator_destroy(f1);
    allocator_destroy(bd);
    return 0;
}

```

Протокол работы программы

./main

```

FreeList alloc time: 0.000053
FreeList usage: 48.83%
FreeList free time: 0.000009
Buddy alloc time: 0.000051
Buddy usage: 97.66%
Buddy free time: 0.000012

```

Strace ./main

```

execve("./main", ["../main"], 0x7ffdd099dde0 /* 27 vars */) = 0

```



```

write(1, "FreeList free time: 0.000042\n", 29FreeList free time: 0.000042
) = 29
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=2807866}) = 0
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=2894173}) = 0
write(1, "Buddy alloc time: 0.000087\n", 27Buddy alloc time: 0.000087
) = 27
write(1, "Buddy usage: 97.66%\n", 20Buddy usage: 97.66%
) = 20
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=3057881}) = 0
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=3101881}) = 0
write(1, "Buddy free time: 0.000044\n", 26Buddy free time: 0.000044
) = 26
munmap(0x7f97579d3000, 65536) = 0
munmap(0x7f97579c3000, 65536) = 0
exit_group(0) = ?
+++ exited with 0 +++

```

Фрагменты вывода strace:

execve("./main", ["./main"], ...) = 0

<Загружает исполняемый файл main>

brk(NULL)

brk(0x6326bd915000)

<Управляет границей кучи процесса (heap)>

mmap(NULL, 65536, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)

mmap(NULL, 65536, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)

<Каждому аллокатору выделяется отдельный пул памяти>

munmap(0x7f97579d3000, 65536)

munmap(0x7f97579c3000, 65536)

<Освобождение памяти аллокаторов>

clock_gettime(CLOCK_PROCESS_CPUTIME_ID, ...)

<Замер CPU-времени для аллокации и освобождения памяти>

write(1, "FreeList alloc time: ...", ...)

write(1, "Buddy free time: ...", ...)

<Ввод-вывод (печать результатов)>

exit_group(0) <Завершение процесса>

Выход

В данной курсовой работе были реализованы и исследованы два пользовательских аллокатора динамической памяти: аллокатор со списком свободных блоков (Free List), использующий стратегию best-fit, и аллокатор двойников (Buddy Allocator). В ходе тестирования оба аллокатора работали поверх заранее выделенного пула памяти размером

2^{16} байт, при этом выполнялось 1000 запросов на выделение блоков фиксированного размера 32 байта.

Результаты эксперимента показали, что время выделения и освобождения памяти для обоих аллокаторов сопоставимо и находится в пределах одной величины, что объясняется малым объёмом теста и отсутствием высокой конкуренции за память. Незначительное преимущество аллокатора со списком свободных блоков при освобождении памяти обусловлено более простой логикой операции free, не требующей рекурсивного поиска и слияния блоков.

Ключевым отличием между аллокаторами стала эффективность использования памяти. Аллокатор со списком свободных блоков продемонстрировал использование около 48% доступного пула памяти. Это связано с внешней фрагментацией, возникающей вследствие дробления памяти на множество свободных блоков различного размера. Несмотря на то, что суммарный объём свободной памяти остаётся значительным, он оказывается распределённым по несмежным участкам, что ограничивает возможность дальнейших выделений.

Аллокатор двойников, напротив, показал использование около 97% доступной памяти. Такой результат достигается благодаря строгой организации памяти в виде блоков степеней двойки и возможности их эффективного слияния при освобождении. В данном тесте размер запроса хорошо соответствует структуре аллокатора, что практически устраняет внешнюю фрагментацию. Однако это достигается ценой внутренней фрагментации, так как каждый запрос обслуживается блоком большего размера, чем требуется приложению, и часть памяти внутри блока остаётся неиспользуемой.

Таким образом, результаты тестирования подтверждают классический компромисс между внутренней и внешней фрагментацией: аллокатор со списком свободных блоков минимизирует внутренние потери памяти, но страдает от внешней фрагментации, тогда как аллокатор двойников эффективно устраняет внешнюю фрагментацию, жертвуя частью памяти внутри выделенных блоков. Выбор подходящего аллокатора зависит от характера нагрузки и требований к использованию памяти.