

Московский Авиационный Институт
(Национальный Исследовательский Университет)
Институт №8 “Компьютерные науки и прикладная математика”
Кафедра №806 “Вычислительная математика и программирование”

**Курсовой проект по курсу
«Операционные системы»**

Группа: М8О-209БВ-24

Студент: Корепанов Иван Алексеевич

Преподаватель: Миронов Е.С.

Оценка: _____

Дата: 14.12.25

Москва, 2025

Постановка задачи

Вариант 17.

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`.

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: алгоритм Мак-Кьюзи-Кэрелса и алгоритм двойников

Общий метод и алгоритм решения

Использованные системные вызовы:

- `mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)` – выделяет непрерывный регион памяти, возвращает указатель на него.
- `munmap(void *addr, size_t length)` – освобождает регион памяти, ранее отображённый через `mmap`.

Рассмотрим подробнее каждый из алгоритмов, а потом сравним их на различных тестах.

Алгоритм Мак-Кьюзи–Кэрелса

Алгоритм Мак-Кьюзи–Кэрелса представляет собой развитие идей аллокаторов, основанных на разбиении памяти по фиксированным классам размеров, и широко применяется в системном программировании, в частности в ядрах семейства BSD. Основная цель данного подхода заключается в уменьшении внутренней фрагментации и накладных расходов при частом выделении и освобождении небольших блоков памяти.

В отличие от аллокатора, использующего метод двойников, Мак-Кьюзик–Кэрелс организует память не как единое дерево блоков, а как набор страниц фиксированного размера. Каждая страница может находиться в одном из нескольких состояний: быть полностью свободной, быть разбитой на блоки одинакового размера или являться частью крупного непрерывного блока памяти. При этом размер страницы обычно совпадает с размером аппаратной страницы памяти, что упрощает управление и повышает эффективность работы с кэшем.

Для обслуживания запросов на выделение памяти аллокатор использует набор классов размеров, значения которых, как правило, являются степенями двойки (например, 16, 32, 64, 128 байт и т.д.). Для каждого класса размеров поддерживается собственный список страниц, внутри которых размещаются блоки данного размера.

Все блоки, находящиеся внутри одной страницы, имеют одинаковый размер, что позволяет не хранить дополнительную служебную информацию для каждого блока.

При поступлении запроса на выделение памяти размером S аллокатор округляет этот размер до ближайшего подходящего класса. Затем производится поиск страницы соответствующего класса, в которой ещё имеются свободные блоки. Если такая страница найдена, один из свободных блоков помечается как занятый и его адрес возвращается пользователю. Если же все страницы данного класса полностью заняты, аллокатор выделяет новую свободную страницу и разбивает её на блоки выбранного размера, после чего один из этих блоков передаётся пользователю.

Особенностью данного алгоритма является то, что информация о занятости отдельных блоков хранится на уровне страницы, например, в виде битовой карты. Благодаря этому освобождение памяти не требует передачи размера блока — по адресу указателя легко определить страницу, которой он принадлежит, а значит и класс размеров. При освобождении блока соответствующий бит в битовой карте очищается, а счётчик свободных блоков страницы увеличивается. Если после освобождения страница становится полностью свободной, она возвращается в общий список свободных страниц и может быть повторно использована для других классов размеров.

Для выделения крупных объёмов памяти, превышающих максимальный поддерживаемый класс размеров, алгоритм Мак-Кьюзи–Кэрелса использует иной механизм. В этом случае выделяется несколько последовательных страниц, которые помечаются как единый крупный блок. При освобождении такого блока все страницы, входящие в него, возвращаются в список свободных страниц.

Сильной стороной аллокатора Мак-Кьюзи–Кэрелса является высокий фактор использования памяти при интенсивной работе с малыми объектами. Отсутствие управляющих заголовков внутри каждого блока и плотное размещение объектов внутри страниц позволяют существенно снизить внутреннюю фрагментацию. Кроме того, операции освобождения памяти выполняются за константное время, так как не требуют поиска соседних блоков и их объединения.

В то же время недостатком данного подхода является потенциальное снижение производительности при выделении памяти. Для поиска свободного блока может потребоваться последовательный просмотр страниц соответствующего класса и поиск свободного элемента внутри страницы. При большом количестве операций выделения и разнообразии размеров блоков это может приводить к увеличению времени работы по сравнению с алгоритмом двойников. Таким образом, аллокатор Мак-Кьюзи–Кэрелса демонстрирует классический компромисс между эффективностью использования памяти и скоростью выделения блоков.

Принцип работы метода двойников:

Все выделяемые блоки имеют размер степени двойки. Размер изначального единого свободного блока степени 2. Когда требуется выделить блок размера N , то находим минимальную степень 2 больше N . Проверяем наличие свободных страниц такого размера, отдаём если есть, если нет, находим минимальный блок большего размера и рекурсивно разделяем его до нужного размера. Получившиеся блоки пополняют списки свободных блоков. При освобождении блока: находим размер блока, проверяем, свободен ли второй блок и если свободен, объединяем их

и рекурсивно проверяем вышестоящий блок. Размер соседнего блока всегда равен размеру освобождаемого.

Код программы

mkc.c

```
#include <stddef.h>
```

```
#include <stdint.h>
```

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

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

```
#define PAGE_SIZE 4096U
```

```
#define MKC_PAGE_FREE 0xFFFFu
```

```
#define MKC_PAGE_LARGE 0xFFFEu
```

```
static const size_t mkc_classes[] = {  
    16, 32, 64, 128, 256, 512, 1024, 2048  
};
```

```
#define MKC_NUM_CLASSES (sizeof(mkc_classes)/sizeof(mkc_classes[0]))
```

```
typedef struct MKC_Page {  
    uint16_t size_class_index;  
    uint16_t free_count;  
    struct MKC_Page* next;  
    uint32_t bitmap[8];  
} MKC_Page;
```

```
typedef struct Allocator {  
    size_t total_size;  
    void* base_data;  
    size_t pages_count;  
  
    MKC_Page* free_pages;  
    MKC_Page* class_pages[MKC_NUM_CLASSES];  
    MKC_Page* large_pages;  
} Allocator;
```

```
static inline MKC_Page* page_at(Allocator* a, size_t i) {
    return (MKC_Page*)((char*)a->base_data + i * PAGE_SIZE);
}
```

```
static inline void bitmap_clear(uint32_t* b) {
    memset(b, 0, sizeof(uint32_t) * 8);
}
```

```
static inline int bitmap_find(uint32_t* b, int limit) {
    for (int i = 0; i < limit; i++)
        if (!(b[i] >> 5] & (1u << (i & 31))))
            return i;
    return -1;
}
```

```
static inline int slots_in_class(int ci) {
    int s = (PAGE_SIZE - sizeof(MKC_Page)) / mkc_classes[ci];
    return s > 256 ? 256 : s;
}
```

```
static inline int class_index(size_t size) {
    for (int i = 0; i < MKC_NUM_CLASSES; i++)
        if (size <= mkc_classes[i])
            return i;
    return -1;
}
```

```
static MKC_Page* take_free(Allocator* a) {
    MKC_Page* p = a->free_pages;
    if (!p) return NULL;
    a->free_pages = p->next;
    p->next = NULL;
    bitmap_clear(p->bitmap);
    return p;
}
```

```
}
```

```
static void put_free(Allocator* a, MKC_Page* p) {  
    p->size_class_index = MKC_PAGE_FREE;  
    p->free_count = 0;  
    bitmap_clear(p->bitmap);  
    p->next = a->free_pages;  
    a->free_pages = p;  
}
```

```
Allocator* createMKCAallocator(size_t size) {  
    if (size < PAGE_SIZE * 2)  
        return NULL;  
    void* mem = mmap(NULL, size,  
        PROT_READ | PROT_WRITE,  
        MAP_PRIVATE | MAP_ANONYMOUS,  
        -1, 0);  
    if (mem == MAP_FAILED)  
        return NULL;  
    Allocator* a = (Allocator*)mem;  
    memset(a, 0, sizeof(Allocator));  
    a->total_size = size;  
    a->base_data = (char*)mem + PAGE_SIZE;  
    a->pages_count = (size - PAGE_SIZE) / PAGE_SIZE;  
    for (size_t i = 0; i < a->pages_count; i++) {  
        MKC_Page* p = page_at(a, i);  
        p->size_class_index = MKC_PAGE_FREE;  
        bitmap_clear(p->bitmap);  
        p->next = a->free_pages;  
        a->free_pages = p;  
    }  
    return a;  
}
```

```

void destroyMKCAllocator(Allocator* a) {
    if (!a) return;
    munmap(a, a->total_size);
}

void* alloc(Allocator* a, size_t size) {
    if (!a || size == 0)
        return NULL;

    int ci = class_index(size);

    if (ci >= 0) {
        MKC_Page* pg = a->class_pages[ci];
        while (pg && pg->free_count == 0)
            pg = pg->next;

        if (!pg) {
            pg = take_free(a);
            if (!pg) return NULL;
            pg->size_class_index = ci;
            pg->free_count = slots_in_class(ci);
            pg->next = a->class_pages[ci];
            a->class_pages[ci] = pg;
        }

        int slot = bitmap_find(pg->bitmap, slots_in_class(ci));
        if (slot < 0) return NULL;

        pg->bitmap[slot >> 5] |= 1u << (slot & 31);
        pg->free_count--;

        return (char*)pg + sizeof(MKC_Page)
            + slot * mkc_classes[ci];
    }
}

```

```
size_t need = (size + PAGE_SIZE - 1) / PAGE_SIZE;
```

```
size_t run = 0, start = 0;
```

```
int found = 0;
```

```
for (size_t i = 0; i < a->pages_count; i++) {
```

```
    MKC_Page* p = page_at(a, i);
```

```
    if (p->size_class_index == MKC_PAGE_FREE) {
```

```
        if (!run) start = i;
```

```
        if (++run == need) {
```

```
            found = 1;
```

```
            break;
```

```
        }
```

```
    } else {
```

```
        run = 0;
```

```
    }
```

```
}
```

```
if (!found) return NULL;
```

```
MKC_Page* first = page_at(a, start);
```

```
first->size_class_index = MKC_PAGE_LARGE;
```

```
first->free_count = need;
```

```
for (size_t i = 1; i < need; i++)
```

```
    page_at(a, start + i)->size_class_index = MKC_PAGE_LARGE;
```

```
MKC_Page** pp = &a->free_pages;
```

```
while (*pp) {
```

```
    MKC_Page* p = *pp;
```

```
    size_t idx = ((char*)p - (char*)a->base_data) / PAGE_SIZE;
```

```
    if (idx >= start && idx < start + need) {
```

```
        *pp = p->next;
```

```
        continue;
```



```

    }

    pp = &p->next;
}

first->next = a->large_pages;
a->large_pages = first;

return (char*)first + sizeof(MKC_Page);
}

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

    size_t off = (char*)ptr - (char*)a->base_data;
    size_t idx = off / PAGE_SIZE;
    if (idx >= a->pages_count)
        return;

    MKC_Page* pg = page_at(a, idx);

    if (pg->size_class_index == MKC_PAGE_LARGE) {
        size_t n = pg->free_count;
        for (size_t i = 0; i < n; i++)
            put_free(a, page_at(a, idx + i));
        return;
    }

    int ci = pg->size_class_index;
    size_t cls = mkc_classes[ci];
    size_t slot =
        ((char*)ptr - ((char*)pg + sizeof(MKC_Page))) / cls;

    if (!(pg->bitmap[slot >> 5] & (1u << (slot & 31))))

```

```

    return;

pg->bitmap[slot >> 5] &= ~(1u << (slot & 31));
pg->free_count++;

if (pg->free_count == slots_in_class(ci)) {
    MKC_Page** pp = &a->class_pages[ci];
    while (*pp) {
        if (*pp == pg) {
            *pp = pg->next;
            break;
        }
        pp = &(*pp)->next;
    }
    put_free(a, pg);
}

size_t get_free_memory(Allocator* a) {
    if (!a) return 0;

    size_t sum = 0;
    for (MKC_Page* p = a->free_pages; p; p = p->next)
        sum += PAGE_SIZE;

    for (size_t i = 0; i < MKC_NUM_CLASSES; i++) {
        MKC_Page* pg = a->class_pages[i];
        while (pg) {
            sum += pg->free_count * mkc_classes[i];
            pg = pg->next;
        }
    }
    return sum;
}

```

buddy.c

```
#define _GNU_SOURCE
```

```
#include <stddef.h>
```

```
#include <stdint.h>
```

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

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

```
#define MIN_ORDER 4
```

```
typedef struct Header {
```

```
    int order;
```

```
    struct Header* next;
```

```
} Header;
```

```
typedef struct {
```

```
    void* base;
```

```
    size_t total_size;
```

```
    int min_order;
```

```
    int max_order;
```

```
    Header** free_lists;
```

```
} BuddyAllocator;
```

```
static inline size_t offset_of(BuddyAllocator* a, Header* h) {
```

```
    return (size_t)((char*)h - (char*)a->base);
```

```
}
```

```
static inline Header* ptr_from_offset(BuddyAllocator* a, size_t off) {
```

```
    return (Header*)((char*)a->base + off);
```

```
}
```

```
static int remove_from_list(Header** list, Header* h) {
```

```
    Header* prev = NULL;
```

```
    Header* cur = *list;
```

```

while (cur) {
    if (cur == h) {
        if (prev) prev->next = cur->next;
        else *list = cur->next;
        return 1;
    }
    prev = cur;
    cur = cur->next;
}
return 0;
}

int createBuddyAllocator(BuddyAllocator* a, size_t size) {
    if (!a || size < (1UL << MIN_ORDER))
        return 0;

    void* mem = mmap(NULL, size,
        PROT_READ | PROT_WRITE,
        MAP_PRIVATE | MAP_ANONYMOUS,
        -1, 0);

    if (mem == MAP_FAILED)
        return 0;

    a->base = mem;
    a->total_size = size;
    a->min_order = MIN_ORDER;

    size_t t = size;
    int mo = 0;
    while (t >= 1) mo++;
    a->max_order = mo;

    int lists = a->max_order - a->min_order + 1;

```

```

a->free_lists = mmap(NULL,
                      lists * sizeof(Header*),
                      PROT_READ | PROT_WRITE,
                      MAP_PRIVATE | MAP_ANONYMOUS,
                      -1, 0);

if (a->free_lists == MAP_FAILED) {
    munmap(mem, size);
    return 0;
}

for (int i = 0; i < lists; i++)
    a->free_lists[i] = NULL;

Header* h = (Header*)a->base;
h->order = a->max_order;
h->next = NULL;
a->free_lists[a->max_order - a->min_order] = h;

return 1;
}

void* buddy_alloc(BuddyAllocator* a, size_t size) {
    if (!a || size == 0)
        return NULL;

    size_t need = size + sizeof(Header);
    size_t blk = 1;
    int order = 0;

    while (blk < need) {
        blk <<= 1;
        order++;
    }

```

```
}
```

```
if (order < a->min_order)
```

```
    order = a->min_order;
```

```
if (order > a->max_order)
```

```
    return NULL;
```

```
int i = order;
```

```
while (i <= a->max_order &&
```

```
    a->free_lists[i - a->min_order] == NULL)
```

```
    i++;
```

```
if (i > a->max_order)
```

```
    return NULL;
```

```
Header* cur = a->free_lists[i - a->min_order];
```

```
a->free_lists[i - a->min_order] = cur->next;
```

```
cur->next = NULL;
```

```
while (i > order) {
```

```
    i--;
```

```
    size_t off = offset_of(a, cur);
```

```
    size_t half = (size_t)1 << i;
```

```
Header* left = cur;
```

```
Header* right = ptr_from_offset(a, off + half);
```

```
left->order = i;
```

```
right->order = i;
```

```
right->next = a->free_lists[i - a->min_order];
```

```
a->free_lists[i - a->min_order] = right;
```

```
cur = left;
```

```

    }

    cur->order = order;
    return (char*)cur + sizeof(Header);
}

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

    Header* cur = (Header*)((char*)ptr - sizeof(Header));
    int order = cur->order;

    while (order < a->max_order) {
        size_t off = offset_of(a, cur);
        size_t buddy_off = off ^ ((size_t)1 << order);
        Header* buddy = ptr_from_offset(a, buddy_off);

        Header** list = &a->free_lists[order - a->min_order];
        if (!remove_from_list(list, buddy))
            break;

        if ((char*)buddy < (char*)cur)
            cur = buddy;

        order++;
        cur->order = order;
    }

    cur->next = a->free_lists[order - a->min_order];
    a->free_lists[order - a->min_order] = cur;
}

size_t buddy_free_memory(BuddyAllocator* a) {

```

```

    if (!a)
        return 0;

    size_t sum = 0;
    for (int o = a->min_order; o <= a->max_order; o++) {
        Header* h = a->free_lists[o - a->min_order];
        while (h) {
            sum += (size_t)1 << o;
            h = h->next;
        }
    }
    return sum;
}

```

```

void destroyBuddyAllocator(BuddyAllocator* a) {
    if (!a)
        return;

    int lists = a->max_order - a->min_order + 1;
    munmap(a->free_lists, lists * sizeof(Header*));
    munmap(a->base, a->total_size);
}

```

test_allocators.c

```
#define _POSIX_C_SOURCE 200112L
```

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

```
#include <stdlib.h>
```

```
#include <time.h>
```

```
#include "buddy.c"
```

```
#include "mkc.c"
```

```
#define MEMORY_SIZE (4 * 1024 * 1024)
```



```
#define NUM_OPERATIONS 100000
```

```
#define MAX_BLOCK_SIZE 128
```

```
double now_sec() {  
    struct timespec ts;  
    clock_gettime(CLOCK_MONOTONIC_RAW, &ts);  
    return ts.tv_sec + ts.tv_nsec * 1e-9;  
}
```

```
void test_buddy() {  
    printf("Buddy\n");  
  
    BuddyAllocator alloc;  
    if (!createBuddyAllocator(&alloc, MEMORY_SIZE)) {  
        printf("init\n");  
        return;  
    }
```

```
void** ptrs = malloc(NUM_OPERATIONS * sizeof(void*));
```

```
double t1 = now_sec();  
for (int i = 0; i < NUM_OPERATIONS; i++) {  
    size_t s = rand() % MAX_BLOCK_SIZE + 1;  
    ptrs[i] = buddy_alloc(&alloc, s);  
}  
double t2 = now_sec();  
printf("alloc: %f s\n", t2 - t1);
```

```
t1 = now_sec();  
for (int i = 0; i < NUM_OPERATIONS; i++)  
    buddy_free(&alloc, ptrs[i]);  
t2 = now_sec();  
printf("free: %f s\n", t2 - t1);
```

```

destroyBuddyAllocator(&alloc);
if (!createBuddyAllocator(&alloc, MEMORY_SIZE)) {
    printf("reinit\n");
    free(ptrs);
    return;
}

size_t total_req = 0;
for (int i = 0; i < NUM_OPERATIONS; i++) {
    size_t s = rand() % MAX_BLOCK_SIZE + 1;
    ptrs[i] = buddy_alloc(&alloc, s);
    if (ptrs[i])
        total_req += s;
}

size_t free_mem = buddy_free_memory(&alloc);
double used = (double)(MEMORY_SIZE - free_mem);
double util = used > 0 ? (double)total_req / used : 0.0;

printf("utilization: %.2f%%\n\n", util * 100.0);

for (int i = 0; i < NUM_OPERATIONS; i++)
    buddy_free(&alloc, ptrs[i]);

destroyBuddyAllocator(&alloc);
free(ptrs);
}

void test_mkc() {
    printf("MKC\n");

    Allocator* a = createMKCAlocator(MEMORY_SIZE);
    if (!a) {
        printf("init\n");
    }
}

```

```

    return;
}

void** ptrs = malloc(NUM_OPERATIONS * sizeof(void*));

double t1 = now_sec();
for (int i = 0; i < NUM_OPERATIONS; i++) {
    size_t s = rand() % MAX_BLOCK_SIZE + 1;
    ptrs[i] = alloc(a, s);
}
double t2 = now_sec();
printf("alloc: %f s\n", t2 - t1);

t1 = now_sec();
for (int i = 0; i < NUM_OPERATIONS; i++)
    free_block(a, ptrs[i]);
t2 = now_sec();
printf("free: %f s\n", t2 - t1);

destroyMKCAllocator(a);

a = createMKCAllocator(MEMORY_SIZE);
if (!a) {
    printf("reinit\n");
    free(ptrs);
    return;
}

size_t total_req = 0;
for (int i = 0; i < NUM_OPERATIONS; i++) {
    size_t s = rand() % MAX_BLOCK_SIZE + 1;
    ptrs[i] = alloc(a, s);
    if (ptrs[i])
        total_req += s;
}

```

```

    }

    size_t free_mem = get_free_memory(a);
    double used = (double)(MEMORY_SIZE - free_mem);
    double util = used > 0 ? (double)total_req / used : 0.0;

    printf("utilization: %.2f%%\n\n", util * 100.0);

    for (int i = 0; i < NUM_OPERATIONS; i++)
        free_block(a, ptrs[i]);

    destroyMKCAllocator(a);
    free(ptrs);
}

int main() {
    srand(1234567);

    test_buddy();
    test_mkc();

    return 0;
}

```

Протокол работы программы

Алгоритм двойников

alloc: 0.007809 с

free: 0.001273 с

utilization: 55.61 %

Мак-Къзи

alloc: 0.334315 с

free: 0.003118 c

utilization: 72.82 %

strace -f ./a.out

execve("./a.out", ["/a.out"], 0x7ffe7bcc75e8 /* 33 vars */) = 0

brk(NULL) = 0x57cd06434000

access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)

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

fstat(3, {st_mode=S_IFREG|0644, st_size=21784, ...}) = 0

mmap(NULL, 21784, PROT_READ, MAP_PRIVATE, 3, 0) = 0x742cc2ad8000

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\3\0>\0\1\0\0\0\260>\2\0\0\0\0"..., 832) =
832

fstat(3, {st_mode=S_IFREG|0755, st_size=1901536, ...}) = 0

mmap(NULL, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x742cc2ad6000

mmap(NULL, 1914496, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3,
0) = 0x742cc2902000

mmap(0x742cc2924000, 1413120, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x22000) = 0x742cc2924000

mmap(0x742cc2a7d000, 323584, PROT_READ,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x17b000) = 0x742cc2a7d000

mmap(0x742cc2acc000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1c9000) = 0x742cc2acc000

mmap(0x742cc2ad2000, 13952, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x742cc2ad2000

close(3) = 0

arch_prctl(ARCH_SET_FS, 0x742cc2ad7540) = 0

mprotect(0x742cc2acc000, 16384, PROT_READ) = 0

mprotect(0x57ccef1b000, 4096, PROT_READ) = 0

mprotect(0x742cc2b08000, 4096, PROT_READ) = 0

munmap(0x742cc2ad8000, 21784) = 0

fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}) = 0

brk(NULL) = 0x57cd06434000

brk(0x57cd06455000) = 0x57cd06455000

write(1, "\320\220\320\273\320\263\320\276\321\200\320\270\321\202\320\274\320\264\320\262\320\276\320\271\320\275\320\270\320\272\320"..., 36Алгоритм
ДВОЙНИКОВ

) = 36

mmap(NULL, 4194304, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x742cc2502000

mmap(NULL, 152, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x742cc2b07000

mmap(NULL, 802816, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x742cc243e000

write(1, "alloc: 0.008341 s\n", 18alloc: 0.008341 s

) = 18

write(1, "free: 0.000583 s\n", 18free: 0.000583 s

) = 18

munmap(0x742cc2b07000, 152) = 0

munmap(0x742cc2502000, 4194304) = 0

mmap(NULL, 4194304, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x742cc2502000

mmap(NULL, 152, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x742cc2b07000

write(1, "utilization: 55.61 %\n\n", 22utilization: 55.61 %

) = 22

munmap(0x742cc2b07000, 152) = 0

munmap(0x742cc2502000, 4194304) = 0

munmap(0x742cc243e000, 802816) = 0

```

write(1,          "\320\234\320\260\320\272-\320\232\321\214\320\267\320\270\n",
16Мак-Къзи
) = 16
mmap(NULL,          4194304,          PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x742cc2502000
brk(0x57cd06518000)          = 0x57cd06518000
write(1, "alloc: 0.515284 s\n", 18alloc: 0.515284 s
)  = 18
write(1, "free: 0.005004 s\n", 18free: 0.005004 s
)  = 18
munmap(0x742cc2502000, 4194304)          = 0
mmap(NULL,          4194304,          PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x742cc2502000
write(1, "utilization: 72.82 %\n\n", 22utilization: 72.82 %

) = 22
munmap(0x742cc2502000, 4194304)          = 0
exit_group(0)          = ?
+++ exited with 0 +++

```

Вывод

В ходе курсового проекта были реализованы два алгоритма аллокации памяти. Оба аллокатора имеют интерфейс аналогичный функциям malloc и free.

По проведенным тестам можно сделать выводы:

Скорость выделения памяти имеет алгоритм двойников. Это можно объяснить тем, что в данном алгоритме хорошо структурирован поиск.

Скорость освобождения памяти также оказался быстрее у алгоритма двойников. Это говорит о том, что операции освобождения и объединения блоков в алгоритме двойников выполняются достаточно эффективно и требуют минимального количества вспомогательных действий.

Аллокатор Мак-Кьюзи–Кэрелса демонстрирует меньшую скорость как при выделении, так и при освобождении памяти. Это связано с необходимостью поиска подходящей страницы нужного класса размеров и работы с битовыми картами.

По показателю utilization алгоритм Мак-Кьюзи–Кэрелса лучше алгоритма двойников на примерно 20%. Это показывает, что алгоритм Мак-Кьюзи–Кэрелса лучше справляется с фрагментацией.

Более простым в использовании и реализации является алгоритм двойников из-за более лёгкой логики работы.

Таким образом, алгоритм двойников превосходит алгоритм Мак-Кьюзи–Кэрелса в скорости выделения и освобождения памяти, а также в простоте использования. Однако алгоритм Мак-Кьюзи–Кэрелса превосходит алгоритм двойников в utilization, что иногда является более важным фактором при выборе алгоритма.