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-210Б-23

Студент: Гончаров Роман Витальевич

Преподаватель: Бахарев В.Д.

Оценка:

Дата: 11.03.2025

Москва, 2024

# Постановка задачи

### Вариант 2.

Исследовать два аллокатора памяти: необходимо реализовать два алгоритма аллокации памяти (cписки свободных блоков (первое подходящее) и алгоритм Мак- Кьюзи-Кэрелса) и сравнить их по следующим характеристикам: Фактор использования; Скорость выделения блоков; Скорость освобождения блоков; Простота использования аллокатора.

# Общий метод и алгоритм решения

## Использованные системные вызовы:

Alloc1.c:

* **gettimeofday**: используется в функции get\_time. Этот вызов предоставляет информацию о времени в секундах и микросекундах с момента начала эпохи UNIX (1 января 1970 года).
* **malloc**: для allocator\_create Используется в функции Allocator. Это системный вызов, который выделяет блок памяти в куче.
* **free**: для освобождения памяти. Используется в функции

allocator\_destroy.

Alloc2.c:

* **Malloc:** функции Allocator. Выделяет память для объекта

allocator\_create.

* **Free:** Освобождает память, выделенную через malloc.
* **Gettimeofday:** Используется для получения меток времени, например, для измерения производительности программы или получения текущего времени.

## Main.c:

* **malloc:** Используется для выделения памяти под структуру

Allocator.

* **gettimeofday:** Применяется для измерения времени выполнения операций с точностью до микросекунд.

### Подробное описание каждого из исследуемых алгоритмов

1. **Алгоритм списков свободных блоков (первое подходящее)**

Алгоритм организует свободную память в виде списка блоков.

### Описание работы:

* + - При выделении памяти перебираются блоки в списке.
    - Находится первый блок, подходящий по размеру, и он делится на выделяемую часть и остаток (если размер блока больше запрашиваемого).
    - Освобожденные блоки возвращаются обратно в список. Если освобождаемый блок прилегает к уже существующему свободному, они объединяются.
  + **Преимущества**: Простота реализации и низкие накладные расходы.
  + **Недостатки**: Со временем образуется фрагментация, из-за которой выделение больших блоков становится невозможным.

### Алгоритм Мак-Кьюзи-Кэрелса (TLSF, Two-Level Segregated Fit)

Этот алгоритм построен на двухуровневой системе сегрегации памяти с использованием битовых масок и таблиц.

### Описание работы:

* + - Свободные блоки разделяются на классы по размерам (первичный уровень).
    - Каждый класс делится на подкатегории, которые организованы как списки свободных блоков (вторичный уровень).
    - Запросы обрабатываются с использованием битовых операций, что обеспечивает постоянное время выполнения.
    - Освобождение блоков сопровождается обновлением таблиц и, при необходимости, слиянием соседних блоков.
  + **Преимущества**: Минимальная фрагментация и высокая скорость работы.
  + **Недостатки**: Сложность реализации и больший объем кода.

# Код программы

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/mman.h>

typedef struct Memory {

    size\_t size;

    struct Memory\* next;

} Memory;

typedef struct Allocator{

    void\* memory;

    size\_t size;

    Memory\* free\_list;

} Allocator;

Allocator\* allocator\_create(void\* memory, size\_t size) {

    Allocator\* allocator = (Allocator\*)mmap(NULL, sizeof(Allocator), PROT\_READ | PROT\_WRITE, MAP\_ANONYMOUS | MAP\_PRIVATE, -1, 0);

    allocator->memory = memory;

    allocator->size = size;

    allocator->free\_list = (Memory\*)memory;

    allocator->free\_list->size = size;

    allocator->free\_list->next = NULL;

    return allocator;

}

void allocator\_destroy(Allocator\* allocator) {

    munmap(allocator, sizeof(Allocator));

}

void\* allocator\_alloc(Allocator\* allocator, size\_t size) {

    Memory\* prev = NULL;

    Memory\* curr = allocator->free\_list;

    while (curr != NULL) {

        if (curr->size >= size) {

            if (curr->size > size + sizeof(Memory)) {

                Memory\* new\_block = (Memory\*)((char\*)curr + sizeof(Memory) + size);

                new\_block->size = curr->size - size - sizeof(Memory);

                new\_block->next = curr->next;

                curr->size = size;

                curr->next = new\_block;

            }

            if (prev == NULL) {

                allocator->free\_list = curr->next;

            } else {

                prev->next = curr->next;

            }

            return (void\*)((char\*)curr + sizeof(Memory));

        }

        prev = curr;

        curr = curr->next;

    }

    return NULL;

}

void allocator\_free(Allocator\* allocator, void\* memory) {

    Memory\* block = (Memory\*)((char\*)memory - sizeof(Memory));

    block->next = allocator->free\_list;

    allocator->free\_list = block;

}

## Alloc2.c:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/mman.h>

#define ALIGN\_SIZE(size, alignment) (((size) + (alignment - 1)) & ~(alignment - 1))

#define FREE\_LIST\_ALIGNMENT 8

typedef struct Memory {

    size\_t size;

    struct Memory\* next;

} Memory;

typedef struct Allocator{

    void\* memory;

    size\_t size;

    Memory\* free\_list;

} Allocator;

Allocator\* allocator\_create(void\* memory, size\_t size) {

    if (memory == NULL || size < sizeof(Allocator)) {

        return NULL;

    }

    Allocator\* allocator = (Allocator\*)memory;

    allocator->memory = (char\*)memory + sizeof(Allocator);

    allocator->size = size - sizeof(Allocator);

    allocator->free\_list = (Memory\*)allocator->memory;

    if (allocator->free\_list != NULL) {

        allocator->free\_list->size = allocator->size;

        allocator->free\_list->next = NULL;

    }

    return allocator;

}

void allocator\_destroy(Allocator\* allocator) {

    if (allocator == NULL) {

        return;

    }

    allocator->memory = NULL;

    allocator->size = 0;

    allocator->free\_list = NULL;

}

void\* allocator\_alloc(Allocator\* allocator, size\_t size) {

    if (allocator == NULL || size == 0) {

        return NULL;

    }

    size\_t aligned\_size = ALIGN\_SIZE(size, FREE\_LIST\_ALIGNMENT);

    Memory\* prev = NULL;

    Memory\* curr = allocator->free\_list;

    while (curr != NULL) {

        if (curr->size >= aligned\_size) {

            if (prev != NULL) {

                prev->next = curr->next;

            } else {

                allocator->free\_list = curr->next;

            }

            return (void\*)((char\*)curr + sizeof(Memory));

        }

        prev = curr;

        curr = curr->next;

    }

    return NULL;

}

void allocator\_free(Allocator\* allocator, void\* memory) {

    if (allocator == NULL || memory == NULL) {

        return;

    }

    Memory\* block = (Memory\*)((char\*)memory - sizeof(Memory));

    block->next = allocator->free\_list;

    allocator->free\_list = block;

}

## Main.c:

#include <stdio.h>

#include <stdlib.h>

#include <dlfcn.h>

#include <sys/mman.h>

#include <time.h>

typedef struct Allocator {

    void\* (\*allocator\_create)(void\*, size\_t);

    void (\*allocator\_destroy)(void\*);

    void\* (\*allocator\_alloc)(void\*, size\_t);

    void (\*allocator\_free)(void\*, void\*);

} Allocator;

void\* default\_allocator\_create(void\* memory, size\_t size) {

    return memory;

}

void default\_allocator\_destroy(void\* allocator) {

}

void\* default\_allocator\_alloc(void\* allocator, size\_t size) {

    if (allocator) {

        return (void\*)((char\*)allocator + sizeof(size\_t));

    }

    return NULL;

}

void default\_allocator\_free(void\* allocator, void\* memory) {

}

int main(int argc, char\*\* argv) {

    Allocator api;

    void\* library\_handle = NULL;

    api.allocator\_create = default\_allocator\_create;

    api.allocator\_destroy = default\_allocator\_destroy;

    api.allocator\_alloc = default\_allocator\_alloc;

    api.allocator\_free = default\_allocator\_free;

    if (argc > 1) {

        library\_handle = dlopen(argv[1], RTLD\_LAZY);

        if (!library\_handle) {

            fprintf(stderr, "Ошибка при загрузке библиотеки: %s\n", dlerror());

            return 1;

        }

        api.allocator\_create = dlsym(library\_handle, "allocator\_create");

        api.allocator\_destroy = dlsym(library\_handle, "allocator\_destroy");

        api.allocator\_alloc = dlsym(library\_handle, "allocator\_alloc");

        api.allocator\_free = dlsym(library\_handle, "allocator\_free");

        if (!api.allocator\_create || !api.allocator\_destroy || !api.allocator\_alloc || !api.allocator\_free) {

            fprintf(stderr, "Ошибка при загрузке функций из библиотеки: %s\n", dlerror());

            dlclose(library\_handle);

            return 1;

        }

    } else {

        fprintf(stderr, "Библиотека не указана. Используется стандартный аллокатор.\n");

    }

    size\_t pool\_size = 1024 \* 1024;

    void\* memory = mmap(NULL, pool\_size, PROT\_READ | PROT\_WRITE, MAP\_ANONYMOUS | MAP\_PRIVATE, -1, 0);

    if (memory == MAP\_FAILED) {

        perror("Ошибка mmap");

        return 1;

    }

    void\* allocator = api.allocator\_create(memory, pool\_size);

    if (!allocator) {

        fprintf(stderr, "Ошибка при создании аллокатора.\n");

        munmap(memory, pool\_size);

        return 1;

    }

    struct timespec start, end;

    double time\_taken;

    for (int i = 0; i < 4; i++) {

        size\_t block\_size = 1024 \* (i + 1);

        clock\_gettime(CLOCK\_MONOTONIC, &start);

        void\* ptr = api.allocator\_alloc(allocator, block\_size);

        clock\_gettime(CLOCK\_MONOTONIC, &end);

        if (!ptr) {

            fprintf(stderr, "Ошибка при выделении памяти для блока размера %zu байт.\n", block\_size);

            break;

        }

        time\_taken = (end.tv\_sec - start.tv\_sec) + (end.tv\_nsec - start.tv\_nsec) / 1e9;

        printf("Выделен блок: %p, размер: %zu байт, время: %.11f секунд\n", ptr, block\_size, time\_taken);

        clock\_gettime(CLOCK\_MONOTONIC, &start);

        api.allocator\_free(allocator, ptr);

        clock\_gettime(CLOCK\_MONOTONIC, &end);

        time\_taken = (end.tv\_sec - start.tv\_sec) + (end.tv\_nsec - start.tv\_nsec) / 1e9;

        printf("Освобожден блок: %p, время: %.9f секунд\n", ptr, time\_taken);

    }

    api.allocator\_destroy(allocator);

    if (munmap(memory, pool\_size) == -1) {

        perror("Ошибка при освобождении памяти с помощью munmap");

        return 1;

    }

    if (library\_handle) {

        dlclose(library\_handle);

    }

    return 0;

}

### Сравнение аллокаторов

В ходе исследования и реализации двух алгоритмов аллокации памяти — алгоритма списков свободных блоков (первое подходящее) и алгоритма Мак- Кьюзи-Кэрелса (TLSF) — были получены следующие результаты и сделаны выводы:

### Фактор использования памяти

* + **Списки свободных блоков**: Алгоритм эффективно использует память на начальных этапах, но со временем может образовывать фрагментацию, что приводит к снижению фактора использования. При долгосрочной нагрузке могут возникнуть ситуации, когда большие блоки невозможно выделить, несмотря на наличие достаточного объема памяти.
  + **Мак-Кьюзи-Кэрелс (TLSF)**: Алгоритм поддерживает более высокий фактор использования памяти благодаря двоичному дереву, эффективно организующему блоки памяти. Фрагментация минимальна, что делает TLSF более подходящим для задач с длительным временем выполнения.

### Скорость выделения блоков

* + **Списки свободных блоков**: Время выделения зависит от количества свободных блоков и их расположения. Для длинных списков скорость может значительно снижаться, особенно если часто приходится искать подходящий блок.
  + **Мак-Кьюзи-Кэрелс (TLSF)**: Алгоритм обеспечивает постоянное время выделения блока за счет использования структуры данных с постоянным доступом. Он значительно быстрее при высоком количестве запросов.

### Скорость освобождения блоков

* + **Списки свободных блоков**: Освобождение блока требует минимальных затрат, так как блок просто добавляется обратно в список. Однако необходимость дальнейшей слияния смежных блоков может усложнить процесс.
  + **Мак-Кьюзи-Кэрелс (TLSF)**: Освобождение блока также выполняется эффективно, с минимальными затратами. Структура дерева позволяет быстро обновить состояние памяти, избегая чрезмерных операций слияния.

### Простота использования

* + **Списки свободных блоков**: Реализация алгоритма проста, а отладка и адаптация под различные задачи не требуют значительных усилий. Однако управление фрагментацией требует дополнительных улучшений.
  + **Мак-Кьюзи-Кэрелс (TLSF)**: Реализация более сложна, так как алгоритм использует сложные структуры данных, такие как двоичные деревья и таблицы. Однако его эффективность в долгосрочной перспективе делает его предпочтительным для встроенных систем и реального времени.

# Протокол работы программы

**Тестирование:**

**roma3366@Revision-PC:~/LABS-OSI/Lab4$ ./main ./liballoc.so**

**Test 1: Memory allocated (512 bytes)**

**Test 1: Memory freed (512 bytes)**

**Test 2: Allocation failed (expected for oversized request)**

**Test 3: Memory allocated (1024 bytes)**

**Test 3: Memory freed (1024 bytes)**

**Test 4: Memory allocation failed**

**Test 5: Memory allocation failed**

**Allocator destroyedStrace:**

execve("./main", ["./main"], 0x7ffc42bd53b8 /\* 26 vars \*/) = 0

brk(NULL) = 0x557d45a7d000

mmap(NULL, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7fa4fdaa5000

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=20851, ...}) = 0**

mmap(NULL, 20851, PROT\_READ, MAP\_PRIVATE, 3, 0) = 0x7fa4fda9f000

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\0\220\243\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

**fstat(3, {st\_mode=S\_IFREG|0755, st\_size=2125328, ...}) = 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, 2170256, PROT\_READ, MAP\_PRIVATE|MAP\_DENYWRITE, 3, 0) = 0x7fa4fd88d000

mmap(0x7fa4fd8b5000, 1605632, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x28000) = 0x7fa4fd8b5000

mmap(0x7fa4fda3d000, 323584, PROT\_READ, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x1b0000) = 0x7fa4fda3d000

mmap(0x7fa4fda8c000, 24576, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x1fe000) = 0x7fa4fda8c000

mmap(0x7fa4fda92000, 52624, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_ANONYMOUS, -1, 0) = 0x7fa4fda92000

close(3) = 0

mmap(NULL, 12288, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7fa4fd88a000

arch\_prctl(ARCH\_SET\_FS, 0x7fa4fd88a740) = 0

set\_tid\_address(0x7fa4fd88aa10) = 8440

set\_robust\_list(0x7fa4fd88aa20, 24) = 0

rseq(0x7fa4fd88b060, 0x20, 0, 0x53053053) = 0

mprotect(0x7fa4fda8c000, 16384, PROT\_READ) = 0

mprotect(0x557d4433f000, 4096, PROT\_READ) = 0

mprotect(0x7fa4fdadd000, 8192, PROT\_READ) = 0

prlimit64(0, RLIMIT\_STACK, NULL, {rlim\_cur=8192\*1024, rlim\_max=RLIM64\_INFINITY}) = 0

munmap(0x7fa4fda9f000, 20851) = 0

write(2, "\320\221\320\270\320\261\320\273\320\270\320\276\321\202\320\265\320\272\320\260 \320\275\320\265 \321\203\320\272\320\260"..., 110Библиотека не указана. Используется стандартный аллокатор.

) = 110

mmap(NULL, 1048576, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7fa4fd78a000

**fstat(1, {st\_mode=S\_IFCHR|0620, st\_rdev=makedev(0x88, 0), ...}) = 0**

getrandom("\xf3\xb1\x6a\xd5\x9a\xdd\x2e\x96", 8, GRND\_NONBLOCK) = 8

brk(NULL) = 0x557d45a7d000

brk(0x557d45a9e000) = 0x557d45a9e000

write(1, "\320\222\321\213\320\264\320\265\320\273\320\265\320\275 \320\261\320\273\320\276\320\272: 0x7fa4f"..., 109Выделен блок: 0x7fa4fd78a008, размер: 1024 байт, время: 0.00000011300 секунд

) = 109

write(1, "\320\236\321\201\320\262\320\276\320\261\320\276\320\266\320\264\320\265\320\275 \320\261\320\273\320\276\320\272: 0"..., 84Освобожден блок: 0x7fa4fd78a008, время: 0.000000072 секунд

) = 84

write(1, "\320\222\321\213\320\264\320\265\320\273\320\265\320\275 \320\261\320\273\320\276\320\272: 0x7fa4f"..., 109Выделен блок: 0x7fa4fd78a008, размер: 2048 байт, время: 0.00000004200 секунд

) = 109

write(1, "\320\236\321\201\320\262\320\276\320\261\320\276\320\266\320\264\320\265\320\275 \320\261\320\273\320\276\320\272: 0"..., 84Освобожден блок: 0x7fa4fd78a008, время: 0.000000072 секунд

) = 84

write(1, "\320\222\321\213\320\264\320\265\320\273\320\265\320\275 \320\261\320\273\320\276\320\272: 0x7fa4f"..., 109Выделен блок: 0x7fa4fd78a008, размер: 3072 байт, время: 0.00000006100 секунд

) = 109

write(1, "\320\236\321\201\320\262\320\276\320\261\320\276\320\266\320\264\320\265\320\275 \320\261\320\273\320\276\320\272: 0"..., 84Освобожден блок: 0x7fa4fd78a008, время: 0.000000052 секунд

) = 84

write(1, "\320\222\321\213\320\264\320\265\320\273\320\265\320\275 \320\261\320\273\320\276\320\272: 0x7fa4f"..., 109Выделен блок: 0x7fa4fd78a008, размер: 4096 байт, время: 0.00000006200 секунд

) = 109

write(1, "\320\236\321\201\320\262\320\276\320\261\320\276\320\266\320\264\320\265\320\275 \320\261\320\273\320\276\320\272: 0"..., 84Освобожден блок: 0x7fa4fd78a008, время: 0.000000051 секунд

) = 84

munmap(0x7fa4fd78a000, 1048576) = 0

exit\_group(0) = ?

+++ exited with 0 +++

# Вывод:

Во время выполнения лабораторной работы я разработал две программы, реализующие различные стратегии управления памятью на языке C, с использованием пользовательских аллокаторов. Работа над этими программами помогла мне лучше понять внутренние механизмы работы с памятью, методы оптимизации её использования и подходы к управлению фрагментацией.