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: Сетраков Ф.С.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 26.12.24

Москва, 2024

Постановка задачи

Цель работы

Приобретение практических навыков в:

1. Создании аллокаторов памяти и их анализу;
2. Создании динамических библиотек и программ, использующие динамические библиотеки.

Задание

Исследовать два аллокатора памяти: необходимо реализовать два алгоритма аллокации памяти и сравнить их по следующим характеристикам:

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

Требуется создать две динамические библиотеки, реализующие два аллокатора, соответственно. Библиотеки загружаются в память с помощью интерфейса ОС (dlopen / LoadLibrary) для работы с динамическими библиотеками. Выбор библиотеки, реализующей аллокатор, осуществляется чтением первого аргумента при запуске программы (argv[1]). Этот аргумент должен содержать путь до динамической библиотеки (относительный или абсолютный). Если аргумент не передан или по переданному пути библиотеки не оказалось, то указатели на функции, реализующие API аллокатора ниже, должны быть присвоены функциям, которые оборачивают системный аллокатор ОС (mmap / VirtualAlloc) в этот API. Эти аварийные оберточные функции должны быть реализованы внутри программы, которая загружает динамические библиотеки (см. пример на GitHub Gist). Каждый аллокатор памяти должен иметь функции аналогичные стандартным функциям malloc и free (realloc, опционально). Перед работой каждый аллокатор инициализируется свободными страницами памяти, выделенными стандартными средствами ядра (mmap / VirtualAlloc). 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. Каждый аллокатор должен обладать следующим интерфейсом:

- Allocator* allocator_create(void *const memory, const size_t size) (инициализация аллокатора на памяти memory размера size);
- void allocator_destroy(Allocator *const allocator)(деинициализация структуры аллокатора);
- void* allocator_alloc(Allocator *const allocator, constsize_t size) (выделение памяти аллокатором памяти размера size);
- void allocator_free(Allocator *const allocator, void *const memory) (возвращает выделенную память аллокатору);

Алгоритм Мак-Кьюзика-Кэрелса

Алгоритм **McKusick-Karels** — это алгоритм управления памятью, разработанный **Маршаллом Кирком Макьюзиком** (Marshall Kirk McKusick) и **Майклом Дж. Карелсом** (Michael J. Karels) для операционной системы **BSD Unix**. Этот алгоритм был предложен как улучшение стандартного подхода к управлению памятью в ядре Unix, чтобы сделать его более эффективным и масштабируемым.

1. **Использование списков свободных блоков**:
    - Память делится на блоки разных размеров, и для каждого размера поддерживается отдельный список свободных блоков.
    - Это позволяет быстро находить блок подходящего размера.
2. **Динамическое разделение и объединение блоков**:
    - Если блок слишком большой для запроса, он разделяется на два меньших блока. Один из них используется, а другой добавляется в соответствующий список свободных блоков.
    - Когда блок освобождается, он объединяется с соседними свободными блоками, чтобы избежать фрагментации.
3. **Иерархия размеров блоков**:
    - Размеры блоков обычно выбираются как степени двойки или в соответствии с другой удобной схемой (например, геометрическая прогрессия).
    - Это упрощает поиск подходящего блока и управление списками.
4. **Эффективное управление памятью**:
    - Алгоритм минимизирует фрагментацию и обеспечивает быстрое выделение и освобождение памяти.

Алгоритм Блоки по 2^n

Алгоритм аллокации памяти **блоками по степеням двойки** (или **2^n**) — это популярный метод управления памятью, который используется для минимизации фрагментации и упрощения процесса выделения и освобождения памяти. Этот подход основан на том, что все блоки памяти имеют размер, равный степени двойки (например, 1, 2, 4, 8, 16, 32, 64 байта и т.д.). Такой метод часто применяется в аллокаторах памяти, таких как **slab allocator**, **buddy allocator** и других.

Основная идея алгоритма

Алгоритм делит всю доступную память на блоки, размеры которых являются степенями двойки. Когда запрашивается память определенного размера, алгоритм находит ближайший больший блок, равный степени двойки, и выделяет его. Если блок слишком большой, он может быть разделен на меньшие блоки (также степени двойки) до тех пор, пока не будет найден блок подходящего размера.

Тестирование

Аллокатор 2^n
Library loaded successfully

Measuring allocation time...

Allocation time for 10,000 allocations: 0.003781 seconds

Measuring free time...

Free time for 10,000 deallocations: 0.003669 seconds

Аллокатор Мак-Кьюзика-Карелса

Library loaded successfully

Measuring allocation time...

Allocation time for 10,000 allocations: 0.006718 seconds

Measuring free time...

Free time for 10,000 deallocations: 0.004116 seconds

mmap

No library path provided, using fallback allocator

Measuring allocation time...

Allocation time for 10,000 allocations: 0.175377 seconds

Measuring free time...

Free time for 10,000 deallocations: 0.089634 seconds

Код программы

mc_cusic_karels/allocator.c

```
#include "allocator.h"

#include <stdlib.h>

#include <string.h>

#include <stdint.h>

#include <math.h>


#define PAGE_SIZE 4096
```

```
typedef struct Block {
	struct Block* next;
} Block;

typedef struct Allocator {
	Block* free_list[32];
	void* memory_start;
	size_t memory_size;
} Allocator;

static size_t next_power_of_two(size_t size) {
	size--;
	size |= size >> 1;
	size |= size >> 2;
	size |= size >> 4;
	size |= size >> 8;
	size |= size >> 16;
	return size + 1;
}

Allocator* allocator_create(void* const memory, const size_t size) {
	Allocator* allocator = (Allocator*)memory;
	if (size < sizeof(Allocator)) return NULL;

	allocator->memory_start = (char*)memory + sizeof(Allocator);
	allocator->memory_size = size - sizeof(Allocator);

	for (size_t i = 0; i < 32; i++) {
	allocator->free_list[i] = NULL;
```

```
        }

        return allocator;
}

void allocator_destroy(Allocator* const allocator) {
        for (size_t i = 0; i < 32; i++) {
        allocator->free_list[i] = NULL;
        }
}

void* allocator_alloc(Allocator* const allocator, const size_t size) {
        size_t block_size = next_power_of_two(size);
        size_t index = (size_t)log2(block_size);

        if (index >= 32) return NULL;

        Block* block = allocator->free_list[index];
        if (block) {
        allocator->free_list[index] = block->next;
        return (void*)block;
        }

        if (allocator->memory_size < block_size) return NULL;

        void* memory = allocator->memory_start;
        allocator->memory_start = (char*)allocator->memory_start + block_size;
        allocator->memory_size -= block_size;
        return memory;
}
```

```
void allocator_free(Allocator* const allocator, void* const memory) {
	if (!memory) return;

	size_t block_size = next_power_of_two((uintptr_t)memory - (uintptr_t)allocator);
	size_t index = (size_t)log2(block_size);

	if (index >= 32) return;

	Block* block = (Block*)memory;
	block->next = allocator->free_list[index];
	allocator->free_list[index] = block;
}
```

two_degree_allocator/allocator.c

```
#include "allocator.h"
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

#define MAX_CLASSES 10
#define PAGE_SIZE 4096

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

typedef struct Allocator {
	Block* free_list[MAX_CLASSES];
	size_t class_sizes[MAX_CLASSES];
```

```
    void* memory_start;
    size_t memory_size;
} Allocator;

static size_t find_class(size_t size, size_t* class_sizes, size_t num_classes) {
    for (size_t i = 0; i < num_classes; i++) {
    if (size <= class_sizes[i]) {
    return i;
    }
    }
    return num_classes;
}

Allocator* allocator_create(void* const memory, const size_t size) {
    Allocator* allocator = (Allocator*)memory;
    if (size < sizeof(Allocator)) return NULL;

    allocator->memory_start = (char*)memory + sizeof(Allocator);
    allocator->memory_size = size - sizeof(Allocator);

    size_t block_size = 16;
    for (size_t i = 0; i < MAX_CLASSES; i++) {
    allocator->class_sizes[i] = block_size;
    allocator->free_list[i] = NULL;
    block_size *= 2;
    }

    return allocator;
}
```

```
void allocator_destroy(Allocator* const allocator) {
	for (size_t i = 0; i < MAX_CLASSES; i++) {
	allocator->free_list[i] = NULL;
	}
}

void* allocator_alloc(Allocator* const allocator, const size_t size) {
	size_t class_index = find_class(size, allocator->class_sizes, MAX_CLASSES);
	if (class_index >= MAX_CLASSES) return NULL;

	Block* block = allocator->free_list[class_index];
	if (block) {
	allocator->free_list[class_index] = block->next;
	return (void*)block;
	}

	size_t block_size = allocator->class_sizes[class_index];
	if (allocator->memory_size < block_size) return NULL;

	void* memory = allocator->memory_start;
	allocator->memory_start = (char*)allocator->memory_start + block_size;
	allocator->memory_size -= block_size;
	return memory;
}

void allocator_free(Allocator* const allocator, void* const memory) {
	if (!memory) return;

	uintptr_t addr = (uintptr_t)memory - (uintptr_t)allocator;
	if (addr >= allocator->memory_size) return;
```

```
        size_t class_index = 0;
        size_t block_size = 0;
        for (; class_index < MAX_CLASSES; class_index++) {
        block_size = allocator->class_sizes[class_index];
        if ((uintptr_t)memory % block_size == 0) {
        break;
        }
        }

        if (class_index >= MAX_CLASSES) return;

        Block* block = (Block*)memory;
        block->next = allocator->free_list[class_index];
        allocator->free_list[class_index] = block;
    }
```

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
#include <sys/mman.h>
#include <stdint.h>
#include <stddef.h>
#include <time.h>

typedef struct Allocator {
    size_t size;
    void *memory;
} Allocator;
```

```
typedef Allocator* (*allocator_create_f)(void *const memory, const size_t size);
typedef void (*allocator_destroy_f)(Allocator *const allocator);
typedef void* (*allocator_alloc_f)(Allocator *const allocator, const size_t size);
typedef void (*allocator_free_f)(Allocator *const allocator, void *const memory);

static allocator_create_f allocator_create = NULL;
static allocator_destroy_f allocator_destroy = NULL;
static allocator_alloc_f allocator_alloc = NULL;
static allocator_free_f allocator_free = NULL;

Allocator* fallback_allocator_create(void *const memory, const size_t size) {
	Allocator *allocator = (Allocator*) mmap(NULL, sizeof(Allocator), PROT_READ |
PROT_WRITE, MAP_PRIVATE | MAP_ANON, -1, 0);
	if (allocator != MAP_FAILED) {
	allocator->size = size;
	allocator->memory = memory;
	}
	return allocator;
}

void fallback_allocator_destroy(Allocator *const allocator) {
	munmap(allocator->memory, allocator->size);
	munmap(allocator, sizeof(Allocator));
}

void* fallback_allocator_alloc(Allocator *const allocator, const size_t size) {
	return mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANON, -1, 0);
}

void fallback_allocator_free(Allocator *const allocator, void *const memory) {
```

```
        munmap(memory, sizeof(memory));
}

void load_allocator_library(const char *path) {
        void *handle = dlopen(path, RTLD_LAZY);
        if (!handle) {
        fprintf(stderr, "Error loading library: %s\n", dlerror());
        return;
        }

        allocator_create = (allocator_create_f) dlsym(handle, "allocator_create");
        allocator_destroy = (allocator_destroy_f) dlsym(handle, "allocator_destroy");
        allocator_alloc = (allocator_alloc_f) dlsym(handle, "allocator_alloc");
        allocator_free = (allocator_free_f) dlsym(handle, "allocator_free");

        if (!allocator_create || !allocator_destroy || !allocator_alloc || !allocator_free) {
        fprintf(stderr, "Error loading functions from library\n");
        dlclose(handle);
        }
}

double measure_time_allocation(Allocator *allocator, size_t alloc_size, int num_allocs) {
        clock_t start = clock();
        for (int i = 0; i < num_allocs; ++i) {
        void *block = allocator_alloc(allocator, alloc_size);
        if (!block) {
        fprintf(stderr, "Allocation failed at iteration %d\n", i);
        break;
        }
        allocator_free(allocator, block);
```

```
        }
        clock_t end = clock();
        return (double)(end - start) / CLOCKS_PER_SEC;
}

double measure_time_free(Allocator *allocator, size_t alloc_size, int num_allocs) {
        void **blocks = malloc(num_allocs * sizeof(void*));
        for (int i = 0; i < num_allocs; ++i) {
        blocks[i] = allocator_alloc(allocator, alloc_size);
        }

        clock_t start = clock();
        for (int i = 0; i < num_allocs; ++i) {
        allocator_free(allocator, blocks[i]);
        }
        clock_t end = clock();
        free(blocks);
        return (double)(end - start) / CLOCKS_PER_SEC;
}

int main(int argc, char **argv) {
        if (argc < 2) {
        printf("No library path provided, using fallback allocator\n");
        allocator_create = (allocator_create_f) fallback_allocator_create;
        allocator_destroy = (allocator_destroy_f) fallback_allocator_destroy;
        allocator_alloc = (allocator_alloc_f) fallback_allocator_alloc;
        allocator_free = (allocator_free_f) fallback_allocator_free;
        } else {
        load_allocator_library(argv[1]);
        if (!allocator_create) {
```

```
printf("Failed to load library, using fallback allocator\n");
allocator_create = (allocator_create_f) fallback_allocator_create;
allocator_destroy = (allocator_destroy_f) fallback_allocator_destroy;
allocator_alloc = (allocator_alloc_f) fallback_allocator_alloc;
allocator_free = (allocator_free_f) fallback_allocator_free;
} else {
printf("Library loaded successfully\n");
}
}

size_t memory_size = 1024 * 1024 * 10;
void *memory = mmap(NULL, memory_size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANON, -1, 0);
if (memory == MAP_FAILED) {
perror("mmap failed");
return 1;
}

Allocator *allocator = allocator_create(memory, memory_size);
if (!allocator) {
fprintf(stderr, "Allocator creation failed\n");
return 1;
}

printf("Measuring allocation time...\n");
double alloc_time = measure_time_allocation(allocator, 128, 10000);
printf("Allocation time for 10,000 allocations: %.6f seconds\n", alloc_time);

printf("Measuring free time...\n");
double free_time = measure_time_free(allocator, 128, 10000);
```

```
    printf("Free time for 10,000 deallocations: %.6f seconds\n", free_time);

    allocator_destroy(allocator);
    munmap(memory, memory_size);

    return 0;
}
```

Протокол работы программы

setrakovfs@osx ~/oslabs/OSLABS/lab4/src $ docker run -it --rm my-cpp-strace-image strace ./a.out /app/two_deg.so

execve("./a.out", ["./a.out", "/app/two_deg.so"], 0x7ffcefcf65d8 /* 4 vars */) = 0

brk(NULL)   = 0x605712f1f000

mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7d6856289000

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=9303, ...}) = 0

mmap(NULL, 9303, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7d6856286000

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"..., 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) = 0x7d6856074000

mmap(0x7d685609c000, 1605632, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7d685609c000

mmap(0x7d6856224000, 323584, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b0000) = 0x7d6856224000

mmap(0x7d6856273000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1fe000) = 0x7d6856273000

mmap(0x7d6856279000, 52624, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7d6856279000

close(3)   = 0

mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7d6856071000

arch_prctl(ARCH_SET_FS, 0x7d6856071740) = 0

set_tid_address(0x7d6856071a10)   = 9

set_robust_list(0x7d6856071a20, 24)   = 0

rseq(0x7d6856072060, 0x20, 0, 0x53053053) = 0

mprotect(0x7d6856273000, 16384, PROT_READ) = 0

mprotect(0x605711034000, 4096, PROT_READ) = 0

mprotect(0x7d68562c1000, 8192, PROT_READ) = 0

prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0

munmap(0x7d6856286000, 9303)   = 0

getrandom("\xa5\x21\xf1\x85\x1d\x1f\xd6\x48", 8, GRND_NONBLOCK) = 8

```
brk(NULL)                               = 0x605712f1f000
brk(0x605712f40000)                     = 0x605712f40000
openat(AT_FDCWD, "/app/two_deg.so", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=15272, ...}) = 0
mmap(NULL, 16400, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7d685606c000
mmap(0x7d685606d000, 4096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1000) =
0x7d685606d000
mmap(0x7d685606e000, 4096, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7d685606e000
mmap(0x7d685606f000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) =
0x7d685606f000
close(3)                                = 0
mprotect(0x7d685606f000, 4096, PROT_READ) = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}) = 0
write(1, "Library loaded successfully\n", 28Library loaded successfully
) = 28
mmap(NULL, 10485760, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7d6855600000
write(1, "Measuring allocation time...\n", 29Measuring allocation time...
) = 29
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=9471261}) = 0
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=13744736}) = 0
write(1, "Allocation time for 10,000 alloc"..., 57Allocation time for 10,000 allocations: 0.004273 seconds
) = 57
write(1, "Measuring free time...\n", 23Measuring free time...
) = 23
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=15611969}) = 0
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=19353493}) = 0
write(1, "Free time for 10,000 deallocatio"..., 53Free time for 10,000 deallocations: 0.003742 seconds
) = 53
munmap(0x7d6855600000, 10485760)        = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

Вывод

В ходе написания данной лабораторной работы я узнал об устройстве аллокаторов. Научился создавать, подключать и использовать динамические библиотеки. Были реализованы два алгоритма аллокации памяти, работающие через один API и подключаемые через динамические библиотеки. В ходе написания данной лабораторной работы я узнал об устройстве аллокаторов. Научился создавать, подключать и использовать динамические библиотеки. Были реализованы два алгоритма аллокации памяти, работающие через один API и подключаемые через динамические библиотеки.