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: Жданович Е.Т.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 09.01.24

Москва, 2024

Постановка задачи

Вариант 2.

Исследовать два аллокатора памяти: необходимо реализовать два алгоритма аллокации памяти (списки 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.

2. 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

Alloc1.c:

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

```c
#include <stddef.h>
#include <sys/time.h>



typedef struct FreeBlock {
    size_t size;
    struct FreeBlock *next;
} FreeBlock;

typedef struct Allocator {
    void *memory;
    size_t size;
    FreeBlock *free_list;
    size_t allocated_memory;
    size_t freed_memory;
} Allocator;

Allocator* allocator_create(void *memory, size_t size);
void* allocator_malloc(Allocator *allocator, size_t size);
void allocator_free(Allocator *allocator, void *ptr);
void allocator_destroy(Allocator *allocator);
double get_time();

Allocator* allocator_create(void *memory, size_t size) {
    if (size < sizeof(FreeBlock)) {
        printf("Error: Not enough memory to create the allocator.\n");
        return NULL;
    }

    FreeBlock *initial_block = (FreeBlock*)memory;
    initial_block->size = size - sizeof(FreeBlock);
    initial_block->next = NULL;

    Allocator *allocator = (Allocator*)malloc(sizeof(Allocator));
    if (!allocator) {
        printf("Error: Failed to allocate memory for Allocator structure.\n");
        return NULL;
    }

    allocator->memory = memory;
    allocator->size = size;
    allocator->free_list = initial_block;
    allocator->allocated_memory = 0;
    allocator->freed_memory = 0;

    printf("Allocator created with size: %zu bytes\n", size);
    return allocator;
}

void* allocator_malloc(Allocator *allocator, size_t size) {
```

```
        if (size == 0) {
            printf("Error: Cannot allocate 0 bytes.\n");
            return NULL;
        }

        size = (size + sizeof(void*) - 1) & ~(sizeof(void*) - 1);

        FreeBlock *prev = NULL;
        FreeBlock *current = allocator->free_list;

        while (current) {
            if (current->size >= size) {
                if (current->size > size + sizeof(FreeBlock)) {
                    FreeBlock *new_block = (FreeBlock*)((char*)current +
sizeof(FreeBlock) + size);
                    new_block->size = current->size - size - sizeof(FreeBlock);
                    new_block->next = current->next;

                    current->size = size;
                    current->next = NULL;

                    if (prev) {
                        prev->next = new_block;
                    } else {
                        allocator->free_list = new_block;
                    }
                } else {
                    if (prev) {
                        prev->next = current->next;
                    } else {
                        allocator->free_list = current->next;
                    }
                }

                allocator->allocated_memory += size;
                printf("Allocated block of size: %zu bytes\n", size);
                return (void*)((char*)current + sizeof(FreeBlock));
            }

            prev = current;
            current = current->next;
        }

        printf("Error: Not enough memory to allocate %zu bytes.\n", size);
        return NULL;
    }

    void allocator_free(Allocator *allocator, void *ptr) {
        if (!ptr) {
            printf("Error: Attempt to free a null pointer.\n");
```

```
            return;
        }

        FreeBlock *block = (FreeBlock*)((char*)ptr - sizeof(FreeBlock));

        allocator->freed_memory += block->size;
        block->next = allocator->free_list;
        allocator->free_list = block;

        printf("Freed block of size: %zu bytes\n", block->size);
    }

    void allocator_destroy(Allocator *allocator) {
        if (allocator) {
            free(allocator);
            printf("Allocator destroyed.\n");
        } else {
            printf("Error: Invalid allocator.\n");
        }
    }

    double get_time() {
        struct timeval tv;
        gettimeofday(&tv, NULL);
        return tv.tv_sec + tv.tv_usec / 1000000.0;
    }

    int main() {
        char memory_pool[1024];
        Allocator *allocator = allocator_create(memory_pool, sizeof(memory_pool));

        void *block1 = allocator_malloc(allocator, 128);
        void *block2 = allocator_malloc(allocator, 256);

        allocator_free(allocator, block1);
        allocator_free(allocator, block2);

        allocator_destroy(allocator);

        return 0;
    }
```

Alloc2.c:

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

```c
#include <stddef.h>
#include <sys/time.h>



typedef struct Block {
    size_t size;
    bool is_free;
    struct Block *next;
} Block;



typedef struct Allocator {
    void *memory;
    size_t size;
    size_t allocated_memory;
    size_t freed_memory;
    Block *free_list;
} Allocator;

Allocator* allocator_create(void *memory, size_t size);
void* allocator_malloc(Allocator *allocator, size_t size);
void allocator_free(Allocator *allocator, void *ptr);
void allocator_destroy(Allocator *allocator);
double get_time();



Allocator* allocator_create(void *memory, size_t size) {
    if (size < sizeof(Block)) {
        printf("Error: Not enough memory to create the allocator.\n");
        return NULL;
    }

    Block *initial_block = (Block *)memory;
    initial_block->size = size - sizeof(Block);
    initial_block->is_free = true;
    initial_block->next = NULL;

    Allocator *allocator = malloc(sizeof(Allocator));
    if (!allocator) {
        printf("Error: Failed to allocate memory for the allocator structure.\n");
        return NULL;
    }

    allocator->memory = memory;
    allocator->size = size;
    allocator->allocated_memory = 0;
    allocator->freed_memory = 0;
    allocator->free_list = initial_block;

    printf("Allocator created with size: %zu bytes\n", size);
```

```
        return allocator;
    }



    void* allocator_malloc(Allocator *allocator, size_t size) {
        if (size == 0) {
            printf("Error: Cannot allocate 0 bytes.\n");
            return NULL;
        }

        size = (size + sizeof(void *) - 1) & ~(sizeof(void *) - 1);

        Block *current = allocator->free_list;
        Block *prev = NULL;

        while (current) {
            if (current->is_free && current->size >= size) {
                if (current->size > size + sizeof(Block)) {
                    Block *new_block = (Block *)((char *)current + sizeof(Block) +
size);
                    new_block->size = current->size - size - sizeof(Block);
                    new_block->is_free = true;
                    new_block->next = current->next;

                    current->size = size;
                    current->next = new_block;
                }

                current->is_free = false;
                allocator->allocated_memory += size;

                printf("Allocated block of size: %zu bytes\n", size);
                return (void *)((char *)current + sizeof(Block));
            }

            prev = current;
            current = current->next;
        }

        printf("Error: Not enough memory to allocate %zu bytes.\n", size);
        return NULL;
    }

    void allocator_free(Allocator *allocator, void *ptr) {
        if (!ptr) {
            printf("Error: Attempt to free a null pointer.\n");
            return;
        }
```

```
        Block *block = (Block *)((char *)ptr - sizeof(Block));

        if (!block->is_free) {
            block->is_free = true;
            allocator->freed_memory += block->size;
            printf("Freed block of size: %zu bytes\n", block->size);

            Block *current = allocator->free_list;
            while (current) {
                if (current->is_free && current->next && current->next->is_free) {
                    current->size += sizeof(Block) + current->next->size;
                    current->next = current->next->next;
                    printf("Merged free blocks. New size: %zu bytes\n", current-
>size);
                }
                current = current->next;
            }
        } else {
            printf("Error: Double free detected.\n");
        }
    }

    void allocator_destroy(Allocator *allocator) {
        if (allocator) {
            free(allocator);
            printf("Allocator destroyed.\n");
        } else {
            printf("Error: Invalid allocator.\n");
        }
    }

    double get_time() {
        struct timeval tv;
        gettimeofday(&tv, NULL);
        return tv.tv_sec + tv.tv_usec / 1000000.0;
    }

    int main() {
        char memory_pool[1024];
        Allocator *allocator = allocator_create(memory_pool, sizeof(memory_pool));

        void *block1 = allocator_malloc(allocator, 128);
        void *block2 = allocator_malloc(allocator, 256);
        void *block3 = allocator_malloc(allocator, 64);

        allocator_free(allocator, block1);
        allocator_free(allocator, block2);
        allocator_free(allocator, block3);

        allocator_destroy(allocator);
```

```
    return 0;
}
```

Main.c:

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



typedef struct FreeBlock {
    size_t size;
    struct FreeBlock *next;
} FreeBlock;



typedef struct Allocator {
    void *memory;
    size_t size;
    FreeBlock *free_list;
    size_t allocated_memory;
    size_t freed_memory;
} Allocator;



Allocator* allocator_create(void *memory, size_t size);
void* allocator_alloc(Allocator *allocator, size_t size);
void allocator_free(Allocator *allocator, void *ptr);
void allocator_destroy(Allocator *allocator);
double get_time();



Allocator* allocator_create(void *memory, size_t size) {
    if (size < sizeof(FreeBlock)) {
        fprintf(stderr, "Error: Not enough memory to create allocator.\n");
        return NULL;
    }

    FreeBlock *initial_block = (FreeBlock*)memory;
    initial_block->size = size - sizeof(FreeBlock);
    initial_block->next = NULL;

    Allocator *allocator = (Allocator*)malloc(sizeof(Allocator));
    if (!allocator) {
```

```
            fprintf(stderr, "Error: Failed to allocate memory for allocator
structure.\n");
            return NULL;
        }

        allocator->memory = memory;
        allocator->size = size;
        allocator->free_list = initial_block;
        allocator->allocated_memory = 0;
        allocator->freed_memory = 0;

        printf("Allocator created with size: %zu bytes\n", size);
        return allocator;
    }



    void* allocator_alloc(Allocator *allocator, size_t size) {
        if (size == 0) {
            fprintf(stderr, "Error: Cannot allocate 0 bytes.\n");
            return NULL;
        }
        size = (size + sizeof(void*) - 1) & ~(sizeof(void*) - 1);

        FreeBlock *prev = NULL;
        FreeBlock *current = allocator->free_list;

        while (current) {
            if (current->size >= size) {
                if (current->size > size + sizeof(FreeBlock)) {
                    FreeBlock *new_block = (FreeBlock*)((char*)current +
sizeof(FreeBlock) + size);
                    new_block->size = current->size - size - sizeof(FreeBlock);
                    new_block->next = current->next;

                    current->size = size;
                    if (prev) {
                        prev->next = new_block;
                    } else {
                        allocator->free_list = new_block;
                    }
                } else {
                    if (prev) {
                        prev->next = current->next;
                    } else {
                        allocator->free_list = current->next;
                    }
                }

                allocator->allocated_memory += size;
                printf("Allocated block of size: %zu bytes\n", size);
```

```
            return (void*)((char*)current + sizeof(FreeBlock));
        }

        prev = current;
        current = current->next;
    }

    fprintf(stderr, "Error: Not enough memory to allocate %zu bytes.\n", size);
    return NULL;
}



void allocator_free(Allocator *allocator, void *ptr) {
    if (!ptr) {
        fprintf(stderr, "Error: Cannot free a NULL pointer.\n");
        return;
    }

    FreeBlock *block = (FreeBlock*)((char*)ptr - sizeof(FreeBlock));
    allocator->freed_memory += block->size;

    block->next = allocator->free_list;
    allocator->free_list = block;

    printf("Freed block of size: %zu bytes\n", block->size);
}



void allocator_destroy(Allocator *allocator) {
    if (allocator) {
        free(allocator);
        printf("Allocator destroyed.\n");
    } else {
        fprintf(stderr, "Error: Invalid allocator.\n");
    }
}

double get_time() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_sec + tv.tv_usec / 1000000.0;
}



int main() {
    char memory_pool[1024];
    Allocator *allocator = allocator_create(memory_pool, sizeof(memory_pool));

    if (!allocator) {
```

```
            return EXIT_FAILURE;
        }

        double start_time, end_time;

        start_time = get_time();
        void *block1 = allocator_alloc(allocator, 128);
        void *block2 = allocator_alloc(allocator, 256);
        end_time = get_time();
        printf("Allocation time: %.6f seconds\n", end_time - start_time);

        start_time = get_time();
        allocator_free(allocator, block1);
        allocator_free(allocator, block2);
        end_time = get_time();
        printf("Free time: %.6f seconds\n", end_time - start_time);

        double usage_factor = (double)allocator->allocated_memory / allocator->size *
100;
        printf("Memory usage factor: %.2f%%\n", usage_factor);

        allocator_destroy(allocator);

        return EXIT_SUCCESS;
    }
```

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:

1. 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.

2. 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.

3. 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.

4. 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:

lizka@LizaAlisa:~/ЛАБЫ_ОС/Лаба4$ ./main ./Alloc1.so

Allocator created with size: 1024 bytes

Allocated block of size: 128 bytes

Allocated block of size: 256 bytes

Allocation time: 0.000004 seconds

Freed block of size: 128 bytes

Freed block of size: 256 bytes

Free time: 0.000003 seconds

Memory usage factor: 37.50%

Allocator destroyed.

lizka@LizaAlisa:~/ЛАБЫ_ОС/Лаба4$ ./main ./Alloc2.so

Allocator created with size: 1024 bytes

Allocated block of size: 128 bytes

Allocated block of size: 256 bytes

Allocation time: 0.000005 seconds

Freed block of size: 128 bytes

Freed block of size: 256 bytes

Free time: 0.000002 seconds

Memory usage factor: 37.50%

Allocator destroyed.

Strace:

lizka@LizaAlisa:~/ЛАБЫ_ОС/Лаба4$ strace -f ./main

execve("./main", ["./main"], 0x7fff109928a8 /* 27 vars */) = 0

brk(NULL) = 0x5642da22d000

mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fef7c258000

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=19163, ...}) = 0

mmap(NULL, 19163, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fef7c253000

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) = 0x7fef7c041000

mmap(0x7fef7c069000, 1605632, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7fef7c069000

mmap(0x7fef7c1f1000, 323584, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b0000) = 0x7fef7c1f1000

mmap(0x7fef7c240000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1fe000) = 0x7fef7c240000

mmap(0x7fef7c246000, 52624, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7fef7c246000 close(3) = 0

mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fef7c03e000

arch_prctl(ARCH_SET_FS, 0x7fef7c03e740) = 0

set_tid_address(0x7fef7c03ea10) = 2150

set_robust_list(0x7fef7c03ea20, 24) = 0

rseq(0x7fef7c03f060, 0x20, 0, 0x53053053) = 0

mprotect(0x7fef7c240000, 16384, PROT_READ) = 0

mprotect(0x56429e75b000, 4096, PROT_READ) = 0

mprotect(0x7fef7c290000, 8192, PROT_READ) = 0

*prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024,

rlim_max=RLIM64_INFINITY}) = 0

munmap(0x7fef7c253000, 19163) = 0

getrandom("\xe8\x32\x99\xb2\xf1\xd7\x52\x10", 8, GRND_NONBLOCK) = 8

brk(NULL) = 0x5642da22d000

brk(0x5642da24e000) = 0x5642da24e000

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

write(1, "Allocator created with size: 102"..., 40 Allocator created with size: 1024 bytes) = 40

write(1, "Allocated block of size: 128 byt"..., 35 Allocated block of size: 128 bytes) = 35

write(1, "Allocated block of size: 256 byt"..., 35 Allocated block of size: 256 bytes) = 35

write(1, "Allocation time: 0.000334 second"..., 34Allocation time: 0.000334 seconds) = 34

write(1, "Freed block of size: 128 bytes\n", 31Freed block of size: 128 bytes) = 31

write(1, "Freed block of size: 256 bytes\n", 31Freed block of size: 256 bytes) = 31

write(1, "Free time: 0.000388 seconds\n", 28Free time: 0.000388 seconds) = 28

write(1, "Memory usage factor: 37.50%\n", 28Memory usage factor: 37.50%) = 28

write(1, "Allocator destroyed.\n", 21Allocator destroyed.) = 21

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.