Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовой проект по курсу

«Операционные системы»

СРАВНЕНИЕ АЛГОРИТМОВ АЛЛОКАЦИИ

Студент:Белоносоов Кирилл Алексеевич
Группа: М8О–208Б–21
Вариант:14
Преподаватель: Соколов Андрей Алексеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2022.

Постановка задачи

Цель работы

1. Приобретение практических навыков в использовании знаний, полученных в течении курса
2. Проведение исследования в выбранной предметной области

Задание

Необходимо спроектировать и реализовать программный прототип в соответствии с выбранным вариантом. Произвести анализ и сделать вывод на основании данных, полученных при работе программного прототипа.

Вариант задания

Исследование 2 аллокаторов памяти: необходимо реализовать два алгоритма аллокации памяти и сравнить их по следующим характеристикам:

1. Фактор использования
2. Скорость выделения блоков
3. Скорость освобождения блоков
4. Простота использования аллокатора

Каждый аллокатор памяти должен иметь функции аналогичные стандартным функциям free и malloc (realloc, опционально). 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.
В отчете необходимо отобразить следующее:

1. Подробное описание каждого из исследуемых алгоритмов
2. Процесс тестирования
3. Обоснование подхода тестирования
4. Результаты тестирования
5. Заключение по проведенной работе

Каждый аллокатор должен обладать следующим интерфейсом (могут быть отличия в зависимости от особенностей алгоритма):

- Allocator* createMemoryAllocator(void *realMemory, size_t memory_size) (создание аллокатора памяти размера memory_size)
- void* alloc(Allocator * allocator, size_t block_size) (выделение памяти при помощи аллокатора размера block_size)
- void* free(Allocator * allocator, void * block) (возвращает выделенную память аллокатору)

Вариант №14:
Необходимо сравнить два алгоритма аллокации: списки свободных блоков (первое
подходящее) и алгоритм Мак-Кьюзи-Кэрелса

Общие сведения о программе

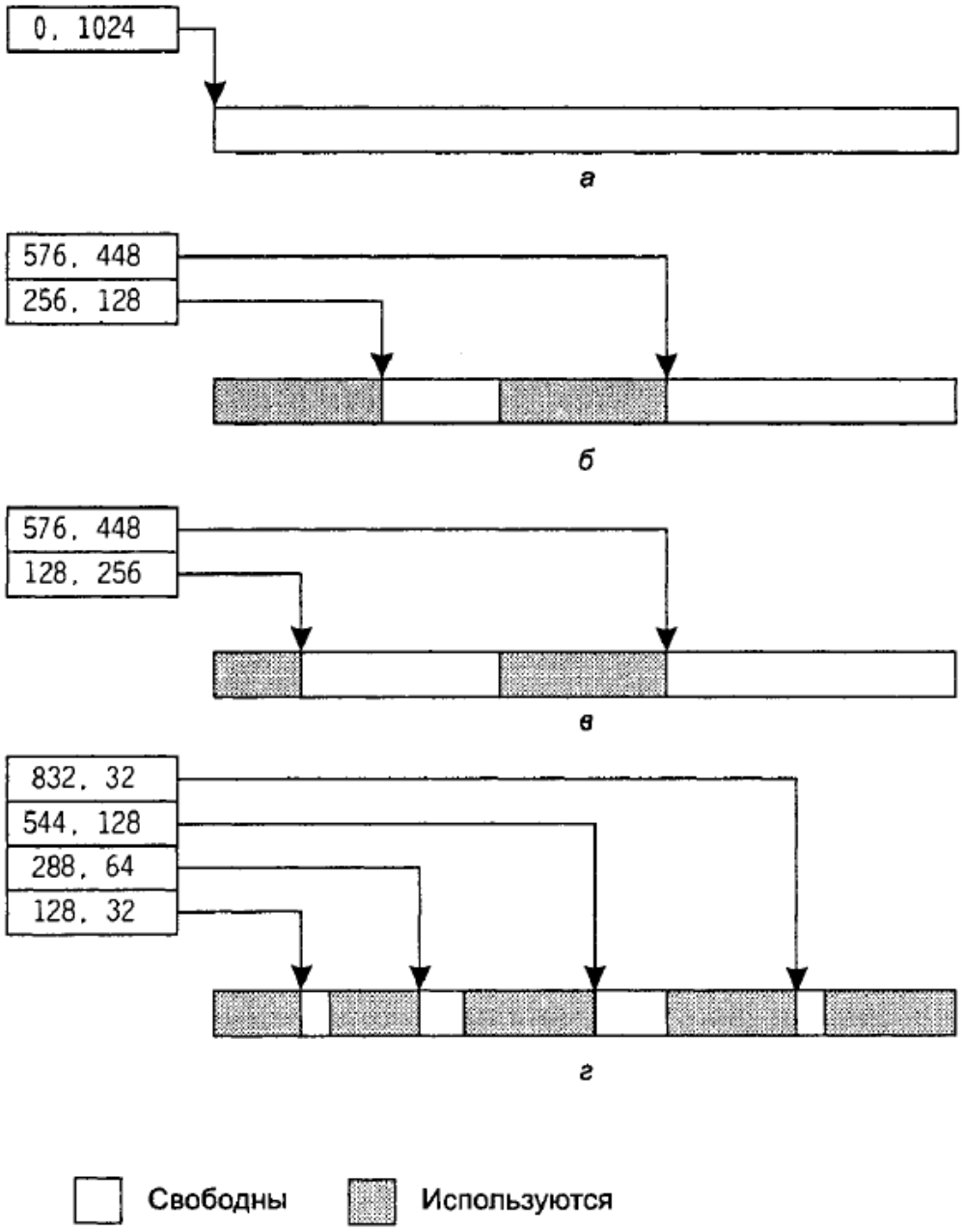
В UNIX оперативная память системы делится на порции фиксированного размера, также называемые страницами. Размер страницы памяти является некоторой степенью числа 2, в большинстве случаев ее объем составляет 4 Кбайт. UNIX является системой виртуальной памяти, поэтому логически последовательные страницы памяти не всегда имеют тот же порядок размещения в физической памяти компьютера.
Менеджер памяти — часть компьютерной программы (как прикладной, так и операционной системы), обрабатывающая запросы на выделение и освобождение оперативной памяти или (для некоторых архитектур ЭВМ) запросы на включение заданной области памяти в адресное пространство процессора.

Основное назначение менеджера памяти в первом смысле — реализация динамической памяти. Например, в языке C динамическое выделение памяти производится через функцию malloc.

Существуют различные алгоритмы для реализации аллокаторов памяти. Все они обладают своими особенностями и недостатками, и каждый из них может быть применим в различных ситуациях. Для данного курсового проекта я реализовал 2 алгоритма:

- Алгоритм с выбором первого подходящего участка памяти, основанный на свободных блоках
- Алгоритм Мак-Кьюзика-Кэрелса

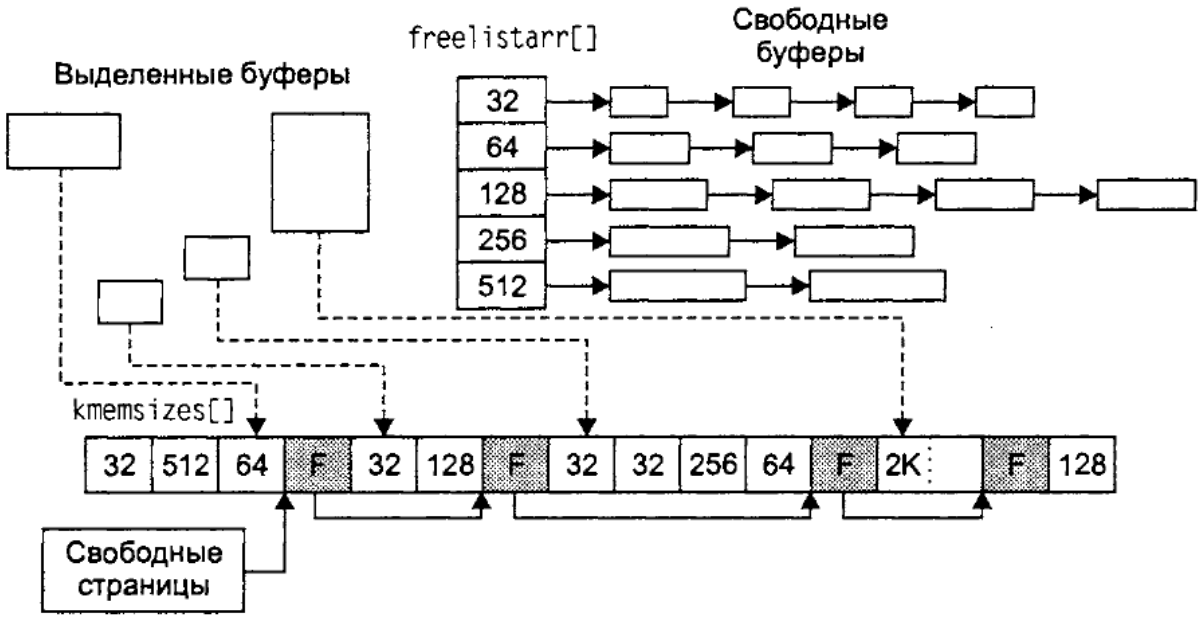
Алгоритм списка свободных блоков



Карта ресурсов (resource map) - это набор пар <base, size> (<базовый адрес, размер>), используемый для отслеживания свободных областей памяти. Изначально область памяти описывается при помощи единственного вхождения карты, в котором указатель равен стартовому адресу области, а размер равен ее общему объему памяти. После этого клиенты начинают запрашивать и освобождать участки памяти, вследствие чего область становится фрагментированной. Ядро создает для каждого нового последовательного свободного участка памяти новое вхождение карты. Элементы карты сортируются в порядке возрастания адресов, что упрощает задачу слияния свободных участков.
При помощи карты ресурсов ядро может выполнить запросы на выделение памяти исходя из трех правил:

- Первый подходящий участок. Выделение памяти из первой по счету свободной области, имеющей достаточный для удовлетворения запроса объем. Это самый быстрый алгоритм из всех трех, но он не совсем оптимален применительно к соображениям уменьшения фрагментации.
- Наиболее подходящий участок. Выделение памяти из области наименьшего размера, достаточного для удовлетворения запроса. Основным недостатком метода является необходимость иногда пропускать некоторое количество сегментов, которые являются слишком мелкими.
- Наименее подходящий участок. Выделение области максимального размера до тех пор, пока не будет найден более подходящий вариант. Такой подход кажется не совсем логичным, однако его применение основано на ожидании, что выделенная область будет достаточно велика для использования в последующих запросах

Алгоритм Мак-Кьюзика-Кэрелса



Маршалл Кирк Мак-Кьюзик и Майкл Дж. Кэрелс разработали усовершенствованный метод выделения памяти, который был реализован во многих вариантах системы UNIX, в том числе 4.4BSD и Digital UNIX. Методика позволяет избавиться от потерь в тех случаях, когда размер запрашиваемого участка памяти равен некоторой степени двойки. В нем также была произведена оптимизация перебора в цикле. Такие действия теперь нужно производить только в том случае, если на момент компиляции неизвестен размер выделенного участка. Алгоритм подразумевает, что память разбита на набор последовательных страниц, и все буферы, относящиеся к одной странице, должны иметь одинаковый размер (являющийся некоторой степенью числа 2). Для управления страницами распределитель использует дополнительный массив kmemsizes[]. Каждая страница может находиться в одном из трех перечисленных состояний:

- Быть свободной. В этом случае соответствующий элемент массива kmemsizes[] содержит указатель на элемент, описывающий следующую свободную страницу.
- Быть разбитой на буферы определенного размера. Элемент массива содержит размер буфера.
- Являться частью буфера, объединяющего сразу несколько страниц. Элемент массива указывает на первую страницу буфера, в которой находятся данные о его длине.

Так как длина всех буферов одной страницы одинакова, нет нужды хранить в заголовках выделенных буферов указатель на список свободных буферов. Процедура free() находит страницу путем маскирования младших разрядов адреса буфера и обнаружения размера буфера в соответствующем элементе массива kmemsizes[]. Отсутствие заголовка в выделенных буферах позволяет экономить память при удовлетворении запросов с потребностью в памяти, кратной некоторой степени числа 2.

Приведенный алгоритм значительно улучшает методику распределения памяти на основе степени числа 2. Он работает намного быстрее, потери памяти при его применении сильно сокращаются. Алгоритм позволяет эффективно обрабатывать запросы на выделения как малых, так и больших участков памяти. Однако описанная методика обладает и некоторыми недостатками, связанными с необходимостью использования участков, равных некоторой степени числа 2. Не существует какого-либо способа перемещения участков из одного списка в другой. Это делает распределитель не совсем подходящим средством при неравномерном использовании памяти, например, если системе необходимо много буферов малого размера на короткий промежуток времени. Технология также не дает возможности возвращать участки памяти, запрошенные ранее у страничной системы.

Общий метод и алгоритм решения.

Для реализации поставленной задачи необходимо:

1. Изучить принципы работы аллокаторов
2. Познакомиться отдельно с каждым из алгоритмов
3. Реализовать алгоритмы и провести отладку
4. Произвести тестирование аллгоритмов

Основные файлы программы

list_alloc.h:

```
#ifndef MYMALLOC_H
#define MYMALLOC_H
```

```
#include <stdlib.h>
#include <stdbool.h>
#define PAGE_SIZE 4096

struct list_block{
   size_t block_size;
   bool is_free;
   struct list_block * next;
   struct list_block * prev;
};

typedef struct list_block list_block;

struct list_Allocator {
   size_t size;
   size_t useful_list;
   size_t total_list;
   void * ptr;
};

typedef struct list_Allocator list_Allocator;

list_Allocator createMemorylist_Allocator(size_t memory_size);
void * list_alloc(list_Allocator * allocator, size_t block_size);
void list_free(void * block);
void list_Alloc_free(list_Allocator * allocator);
size_t get_useful_list(list_Allocator * allocator);
size_t get_total_list(list_Allocator * allocator);
#endif
```

list_alloc.c

```
#include "list_alloc.h"
```

```
#include <errno.h>
#include <stdio.h>
#include <sys/mman.h>
list_Allocator createMemorylist_Allocator(size_t memory_size) {
    size_t pages = memory_size / PAGE_SIZE + ((memory_size % PAGE_SIZE != 0) ? 1 : 0);
    size_t size = pages * PAGE_SIZE;
    void * realMemory = mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_PRIVATE|
MAP_ANONYMOUS, -1, 0);
    if(realMemory == MAP_FAILED) {
        perror("Error: list_Allocator not enough memory");
        exit(EXIT_FAILURE);
    }
    list_block * start_list = (list_block *)realMemory;
    start_list->next = NULL;
    start_list->prev = NULL;
    start_list->block_size = size - sizeof(list_block);
    start_list->is_free = true;
    return (list_Allocator){.size=size, .total_list=0, .useful_list=0, .ptr=realMemory};
}

void * list_alloc(list_Allocator * allocator, size_t block_size) {
    if(allocator == NULL)
        return NULL;
    if(block_size == 0)
        return NULL;
    size_t new_size = 0;
    if(block_size % 4 != 0) {
        new_size = block_size + block_size % 4;
    } else {
        new_size = block_size;
    }
    if(allocator->size < new_size)
        return NULL;
    list_block * list = (list_block *)allocator->ptr;
```

```
    void * ptr = NULL;
    while(list != NULL) {
        if(list->block_size >= new_size + sizeof(list_block) && list->is_free) {
            list->next = (list_block *)((void *)list + sizeof(list_block) + new_size);
            list->next->prev = list;
            list->next->block_size = list->block_size - new_size - sizeof(list_block);
            list->next->is_free = true;
            list->block_size = new_size;
            list->is_free = false;
            ptr = (void *)list + sizeof(list_block);
            break;
        } else {
            list = list->next;
        }
    }
    if(ptr != NULL) {
        allocator->useful_list += block_size;
        allocator->total_list += new_size + sizeof(list_block);
    }
    return ptr;
}

void list_free(void * block) {
    list_block * list = (list_block *)(block - sizeof(list_block));
    list->is_free = true;
    if(list->prev != NULL) {
        if(list->prev->is_free) {
            list->prev->block_size = list->prev->block_size + sizeof(list_block) + list->block_size;
            list->prev->next = list->next;
            list = list->prev;
        }
    }
    if(list->next != NULL) {
```

```
        if(list->next->is_free) {
            list->block_size = list->block_size + sizeof(list_block) + list->next->block_size;
            list->next = list->next->next;
        }
    }
}

void list_Alloc_free(list_Allocator * allocator) {
    if(munmap(allocator->ptr, allocator->size) == -1) {
        perror("Error: free list_Allocator is not success");
        exit(EXIT_FAILURE);
    }
}

size_t get_useful_list(list_Allocator * allocator) {
    return allocator->useful_list;
}
size_t get_total_list(list_Allocator * allocator) {
    return allocator->total_list;
}
```

mkk_alloc.h

```
#ifndef MKK_ALLOC_H
#define MKK_ALLOC_H

#include <stdlib.h>
#include <stdbool.h>
#define PAGE_SIZE 4096
#define MIN_BUFFER 4
#define MIN_ALLOC_SIZE (1 << MIN_BUFFER)
#define MAXKMEM (PAGE_SIZE * 1024)

enum States{
    FREE,
```

```
    CONT
};

struct kmem {
    size_t bufindex;
    size_t cnt_pages;
};

typedef struct kmem kmem;

static kmem kmemsizes[MAXKMEM / PAGE_SIZE];

struct kmembucket {
    void * next;
};
typedef struct kmembucket kmembucket;

static kmembucket bucket[MIN_BUFFER + 16];

struct kmem_Allocator {
    size_t size;
    size_t useful_kmem;
    size_t total_kmem;
    void * ptr;
};

typedef struct kmem_Allocator kmem_Allocator;
kmem_Allocator createMemorykmem_Allocator(size_t memory_size);
void * kmem_alloc(kmem_Allocator * allocator, size_t block_size);
void kmem_free(kmem_Allocator * allocator, void * block);
void kmem_Alloc_free(kmem_Allocator * allocator);
size_t get_useful_kmem(kmem_Allocator * allocator);
```

```
size_t get_total_kmem(kmem_Allocator * allocator);
#endif
```

mkk_alloc.c

```
#include "mkk_alloc.h"
#include <errno.h>
#include <stdio.h>
#include <sys/mman.h>
#define NDX(size) \
   (size) <= (MIN_ALLOC_SIZE * 16) \
      ? (size) <= (MIN_ALLOC_SIZE * 4) \
         ? (size) <= (MIN_ALLOC_SIZE * 2) \
            ? (size) <= (MIN_ALLOC_SIZE * 1) \
               ? (MIN_BUFFER + 0) \
               : (MIN_BUFFER + 1) \
            : (MIN_BUFFER + 2) \
         : (size) <= (MIN_ALLOC_SIZE * 8) \
            ? (MIN_BUFFER + 3) \
            : (MIN_BUFFER + 4) \
      : (size) <= (MIN_ALLOC_SIZE * 64) \
         ? (size) <= (MIN_ALLOC_SIZE * 32) \
            ? (MIN_BUFFER + 5) \
            : (MIN_BUFFER + 6) \
         : (size) <= (MIN_ALLOC_SIZE * 128) \
            ? (MIN_BUFFER + 7) \
            : (MIN_BUFFER + 8) \


kmem_Allocator createMemorykmem_Allocator(size_t memory_size) {
   size_t pages = memory_size / PAGE_SIZE + ((memory_size % PAGE_SIZE != 0) ? 1 : 0);
   size_t size = pages * PAGE_SIZE;
   void * ptr = mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_PRIVATE|
MAP_ANONYMOUS, -1, 0);
   if(ptr == MAP_FAILED) {
      perror("Error: kmem_Allocator not enough memory");
```

```
        exit(EXIT_FAILURE);
    }
    for(int i = 0; i < pages; ++i) {
        kmemsizes[i].bufindex = FREE;
    }
    return (kmem_Allocator){.ptr=ptr, .total_kmem=0, .useful_kmem=0, .size=size};
}

void * alloc_page(kmem_Allocator * allocator, size_t block_size) {
    size_t count_of_blocks = PAGE_SIZE / block_size;
    size_t i = 0;
    while(kmemsizes[i].bufindex != FREE) {
        ++i;
    }
    if(i > allocator->size / PAGE_SIZE - 1)
        return NULL;
    kmemsizes[i].bufindex = block_size;
    void * start = (allocator->ptr + i * PAGE_SIZE);
    kmembucket * kbp = &bucket[NDX(block_size)];
    kbp->next = start;
    for (int j = 0; j < count_of_blocks - 1; ++j) {
        kmembucket * cur = (kmembucket *)start;
        cur->next = start + block_size;
        start = cur->next;
    }
    kmembucket * end = (kmembucket *)(start);
    end->next = NULL;
    void * res = kbp->next;
    kbp->next = ((kmembucket *)res)->next;
    return res;
}

void * alloc_pages(kmem_Allocator * allocator, size_t pages) {
```

```
    size_t count_pages = 0;
    size_t i = 0;
    while(i < (allocator->size / PAGE_SIZE) && count_pages != pages) {
        if(kmemsizes[i].bufindex == FREE) {
            ++count_pages;
        }
        else{
            count_pages = 0;
        }
        ++i;
    }
    if(count_pages != pages || i > (allocator->size / PAGE_SIZE)) {
        return NULL;
    }
    void * ptr = allocator->ptr + (i - pages) * PAGE_SIZE;
    kmemsizes[i - pages].bufindex = pages * PAGE_SIZE;
    for(int j = i - pages + 1; j != i; ++j) {
        kmemsizes[j].bufindex = CONT;
    }
    return ptr;
}

void * kmem_alloc(kmem_Allocator * allocator, size_t block_size) {
    if(allocator == NULL)
        return NULL;
    if(block_size == 0)
        return NULL;
    void * ptr = NULL;
    size_t total_size = 0;
    if(block_size <= PAGE_SIZE / 2) {
        kmembucket * kbp = &bucket[NDX(block_size)];
        if(kbp->next == NULL) {
            ptr = alloc_page(allocator, 1 << (NDX(block_size) - MIN_BUFFER));
```

```
        } else {
            ptr = kbp->next;
            kbp->next = ((kmembucket *)ptr)->next;
        }
        total_size = 1 << (NDX(block_size) - MIN_BUFFER);
    }
    else {
        size_t pages = block_size / PAGE_SIZE + ((block_size % PAGE_SIZE != 0) ? 1 : 0);
        ptr = alloc_pages(allocator, pages);
        total_size = pages * PAGE_SIZE;
    }
    if(ptr != NULL) {
        allocator->useful_kmem += block_size;
        allocator->total_kmem += total_size;
    }
    return ptr;
}

void kmem_free(kmem_Allocator * allocator, void * block) {
    kmem * ksp = &kmemsizes[(block - allocator->ptr) / PAGE_SIZE];
    kmembucket * kbp;
    if(ksp->bufindex <= PAGE_SIZE / 2) {
        kbp = &bucket[NDX(ksp->bufindex)];
        ((kmembucket *)block)->next = kbp->next;
        kbp->next = block;
    }
    else {
        size_t pages = ksp->bufindex / PAGE_SIZE;
        ksp->bufindex = FREE;
        for(int i = 1; i < pages - 1; ++i) {
            kmemsizes[(block - allocator->ptr) / PAGE_SIZE].bufindex = FREE;
        }
    }
```

```
}
void kmem_Alloc_free(kmem_Allocator * allocator) {
    if(munmap(allocator->ptr, allocator->size) == -1) {
        perror("Error: free kmem_Allocator is not success");
        exit(EXIT_FAILURE);
    }
}

size_t get_useful_kmem(kmem_Allocator * allocator) {
    return allocator->useful_kmem;
}
size_t get_total_kmem(kmem_Allocator * allocator) {
    return allocator->total_kmem;
}
```

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <errno.h>
#include "include/list_alloc.h"
#include "include/mkk_alloc.h"

typedef struct
{
    void * addr;
    size_t size;
} data_alloc;



const size_t malloc_request = 100000;
const size_t first_malloc_lower_limit = 4;
const size_t first_malloc_upper_limit = 2048;
```

```
const size_t second_malloc_lower_limit = 4;
const size_t second_malloc_upper_limit = 81920;
const size_t third_malloc_lower_limit = 4;
const size_t third_malloc_upper_limit = 4096000;

int main(int argc, char * argv[]) {
    clock_t start_time = 0;
    clock_t end_time = 0;
    size_t useful = 0;
    size_t total = 0;
    if(argc < 2) {
        fprintf(stderr, "Error: input test grade and size of heap\n");
        exit(EXIT_FAILURE);
    }
    srand((unsigned int)time(0));

    size_t heap_size = strtoul(argv[1], NULL, 0);
    list_Allocator list_allocator = createMemorylist_Allocator(heap_size);
    kmem_Allocator kmem_allocator = createMemorykmem_Allocator(heap_size);

    data_alloc first_m_requests[malloc_request];
    data_alloc second_m_requests[malloc_request];
    data_alloc third_m_requests[malloc_request];

    for(int i = 0; i < malloc_request; ++i) {
        first_m_requests[i].size = first_malloc_lower_limit + rand() % first_malloc_upper_limit;
    }
    printf("LIST ALLOCATOR LOWER TEST: \n");

    start_time = clock();

    for(int i = 0; i < malloc_request; ++i) {
        first_m_requests[i].addr = list_alloc(&list_allocator, first_m_requests[i].size);
```

```
}

end_time = clock();

printf("Requests: %ld\n", malloc_request);
printf("Heap size: %ld bytes\n", heap_size);
printf("Limits: %ld to %ld bytes\n", first_malloc_lower_limit, first_malloc_upper_limit);
printf("Allocation time: %lfs\n",(double)(end_time - start_time)/CLOCKS_PER_SEC);

for(int i = 0; i < malloc_request / 2; ++i) {
    int j = rand() % (malloc_request - 1);
    data_alloc t = first_m_requests[i];
    first_m_requests[i] = first_m_requests[j];
    first_m_requests[j] = t;
}

start_time = clock();

for(int i = 0; i < malloc_request; ++i) {
    if(first_m_requests[i].addr != NULL) {
        list_free(first_m_requests[i].addr);
    }
}

end_time = clock();

useful = list_allocator.useful_list;
total = list_allocator.total_list;
printf("free time: %lfs\n",(double)(end_time - start_time)/CLOCKS_PER_SEC);
printf("Usage: %lf%%\n\n", (double)useful / total);

for(int i = 0; i < malloc_request; ++i) {
```

```
    first_m_requests[i].size = first_malloc_lower_limit + rand() % first_malloc_upper_limit;
}

printf("KMEM ALLOCATOR LOWER TEST: \n");

start_time = clock();

for(int i = 0; i < malloc_request; ++i) {
    first_m_requests[i].addr = kmem_alloc(&kmem_allocator, first_m_requests[i].size);
}

end_time = clock();

printf("Requests: %ld\n", malloc_request);
printf("Heap size: %ld bytes\n", heap_size);
printf("Limits: %ld to %ld bytes\n", first_malloc_lower_limit, first_malloc_upper_limit);
printf("Allocation time: %lfs\n",(double)(end_time - start_time)/CLOCKS_PER_SEC);

for(int i = 0; i < malloc_request / 2; ++i) {
    int j = rand() % (malloc_request - 1);
    data_alloc t = first_m_requests[i];
    first_m_requests[i] = first_m_requests[j];
    first_m_requests[j] = t;
}

start_time = clock();

for(int i = 0; i < malloc_request; ++i) {
    if(first_m_requests[i].addr != NULL) {
        kmem_free(&kmem_allocator, first_m_requests[i].addr);
    }
}
```

```
    end_time = clock();

    useful = kmem_allocator.useful_kmem;
    total = kmem_allocator.total_kmem;
    printf("free time: %lfs\n",(double)(end_time - start_time)/CLOCKS_PER_SEC);
    printf("Usage: %lf%%\n\n", (double)useful / total);

    list_Alloc_free(&list_allocator);
    kmem_Alloc_free(&kmem_allocator);

    list_allocator = createMemorylist_Allocator(heap_size);
    kmem_allocator = createMemorykmem_Allocator(heap_size);

    for(int i = 0; i < malloc_request; ++i) {
        second_m_requests[i].size = second_malloc_lower_limit + rand() %
second_malloc_upper_limit;
    }

    printf("LIST ALLOCATOR MIDDLE TEST: \n");

    start_time = clock();

    for(int i = 0; i < malloc_request; ++i) {
        second_m_requests[i].addr = list_alloc(&list_allocator, second_m_requests[i].size);
    }

    end_time = clock();

    printf("Requests: %ld\n", malloc_request);
    printf("Heap size: %ld bytes\n", heap_size);
    printf("Limits: %ld to %ld bytes\n", second_malloc_lower_limit,
second_malloc_upper_limit);
```

```
    printf("Allocation time: %lfs\n",(double)(end_time - start_time)/CLOCKS_PER_SEC);

    for(int i = 0; i < malloc_request / 2; ++i) {
        int j = rand() % (malloc_request - 1);
        data_alloc t = second_m_requests[i];
        second_m_requests[i] = second_m_requests[j];
        second_m_requests[j] = t;
    }

    start_time = clock();

    for(int i = 0; i < malloc_request; ++i) {
        if(second_m_requests[i].addr != NULL) {
            list_free(second_m_requests[i].addr);
        }
    }

    end_time = clock();

    useful = list_allocator.useful_list;
    total = list_allocator.total_list;
    printf("free time: %lfs\n",(double)(end_time - start_time)/CLOCKS_PER_SEC);
    printf("Usage: %lf%%\n\n", (double)useful / total);

    for(int i = 0; i < malloc_request; ++i) {
        second_m_requests[i].size = second_malloc_lower_limit + rand() %
second_malloc_upper_limit;
    }

    printf("KMEM ALLOCATOR MIDDLE TEST: \n");

    start_time = clock();
```

```
    for(int i = 0; i < malloc_request; ++i) {
        second_m_requests[i].addr = kmem_alloc(&kmem_allocator, second_m_requests[i].size);
    }

    end_time = clock();

    printf("Requests: %ld\n", malloc_request);
    printf("Heap size: %ld bytes\n", heap_size);
    printf("Limits: %ld to %ld bytes\n", second_malloc_lower_limit,
second_malloc_upper_limit);
    printf("Allocation time: %lfs\n",(double)(end_time - start_time)/CLOCKS_PER_SEC);

    for(int i = 0; i < malloc_request / 2; ++i) {
        int j = rand() % (malloc_request - 1);
        data_alloc t = second_m_requests[i];
        second_m_requests[i] = second_m_requests[j];
        second_m_requests[j] = t;
    }

    start_time = clock();

    for(int i = 0; i < malloc_request; ++i) {
        if(second_m_requests[i].addr != NULL) {
            kmem_free(&kmem_allocator, second_m_requests[i].addr);
        }
    }

    end_time = clock();

    useful = kmem_allocator.useful_kmem;
    total = kmem_allocator.total_kmem;
    printf("free time: %lfs\n",(double)(end_time - start_time)/CLOCKS_PER_SEC);
    printf("Usage: %lf%%\n\n", (double)useful / total);
```

```
list_Alloc_free(&list_allocator);
kmem_Alloc_free(&kmem_allocator);

list_allocator = createMemorylist_Allocator(heap_size);
kmem_allocator = createMemorykmem_Allocator(heap_size);

for(int i = 0; i < malloc_request; ++i) {
   third_m_requests[i].size = third_malloc_lower_limit + rand() % third_malloc_upper_limit;
}

printf("LIST ALLOCATOR UPPER TEST: \n");

start_time = clock();

for(int i = 0; i < malloc_request; ++i) {
   third_m_requests[i].addr = list_alloc(&list_allocator, third_m_requests[i].size);
}

end_time = clock();

printf("Requests: %ld\n", malloc_request);
printf("Heap size: %ld bytes\n", heap_size);
printf("Limits: %ld to %ld bytes\n", third_malloc_lower_limit, third_malloc_upper_limit);
printf("Allocation time: %lfs\n",(double)(end_time - start_time)/CLOCKS_PER_SEC);

for(int i = 0; i < malloc_request / 2; ++i) {
   int j = rand() % (malloc_request - 1);
   data_alloc t = third_m_requests[i];
   third_m_requests[i] = third_m_requests[j];
   third_m_requests[j] = t;
}
```

```
start_time = clock();

for(int i = 0; i < malloc_request; ++i) {
    if(third_m_requests[i].addr != NULL) {
        list_free(third_m_requests[i].addr);
    }
}

end_time = clock();

useful = list_allocator.useful_list;
total = list_allocator.total_list;
printf("free time: %lfs\n",(double)(end_time - start_time)/CLOCKS_PER_SEC);
printf("Usage: %lf%%\n\n", (double)useful / total);

for(int i = 0; i < malloc_request; ++i) {
    third_m_requests[i].size = third_malloc_lower_limit + rand() % third_malloc_upper_limit;
}

printf("KMEM ALLOCATOR UPPER TEST: \n");

start_time = clock();

for(int i = 0; i < malloc_request; ++i) {
    third_m_requests[i].addr = kmem_alloc(&kmem_allocator, third_m_requests[i].size);
}

end_time = clock();

printf("Requests: %ld\n", malloc_request);
printf("Heap size: %ld bytes\n", heap_size);
```

```
    printf("Limits: %ld to %ld bytes\n", third_malloc_lower_limit, third_malloc_upper_limit);
    printf("Allocation time: %lfs\n",(double)(end_time - start_time)/CLOCKS_PER_SEC);

    for(int i = 0; i < malloc_request / 2; ++i) {
        int j = rand() % (malloc_request - 1);
        data_alloc t = third_m_requests[i];
        third_m_requests[i] = third_m_requests[j];
        third_m_requests[j] = t;
    }

    start_time = clock();

    for(int i = 0; i < malloc_request; ++i) {
        if(third_m_requests[i].addr != NULL) {
            kmem_free(&kmem_allocator, third_m_requests[i].addr);
        }
    }

    end_time = clock();

    useful = kmem_allocator.useful_kmem;
    total = kmem_allocator.total_kmem;
    printf("free time: %lfs\n",(double)(end_time - start_time)/CLOCKS_PER_SEC);
    printf("Usage: %lf%%\n\n", (double)useful / total);

    list_Alloc_free(&list_allocator);
    kmem_Alloc_free(&kmem_allocator);
    return 0;
}
```

Пример работы

kirill@kirill:~/localProjects/OSlabs/kp/build/src$
/home/kirill/localProjects/OSlabs/kp/build/src/main 10000

LIST ALLOCATOR LOWER TEST:

Requests: 100000

Heap size: 10000 bytes

Limits: 4 to 2048 bytes

Allocation time: 0.007658s

free time: 0.000493s

Usage: 0.964000%

KMEM ALLOCATOR LOWER TEST:

Requests: 100000

Heap size: 10000 bytes

Limits: 4 to 2048 bytes

Allocation time: 0.004223s

free time: 0.000541s

Usage: 0.759684%

LIST ALLOCATOR MIDDLE TEST:

Requests: 100000

Heap size: 10000 bytes

Limits: 4 to 81920 bytes

Allocation time: 0.002126s

free time: 0.000696s

Usage: 0.986275%

KMEM ALLOCATOR MIDDLE TEST:

Requests: 100000

Heap size: 10000 bytes

Limits: 4 to 81920 bytes

Allocation time: 0.005009s

free time: 0.000625s

Usage: 0.805261%

LIST ALLOCATOR UPPER TEST:

Requests: 100000

Heap size: 10000 bytes

Limits: 4 to 4096000 bytes

Allocation time: 0.001227s

free time: 0.000358s

Usage: 0.977803%

KMEM ALLOCATOR UPPER TEST:

Requests: 100000

Heap size: 10000 bytes

Limits: 4 to 4096000 bytes

Allocation time: 0.002097s

free time: 0.000336s

Usage: 0.766113%

kirill@kirill:~/localProjects/OSlabs/kp/build/src$
/home/kirill/localProjects/OSlabs/kp/build/src/main 100000

LIST ALLOCATOR LOWER TEST:

Requests: 100000

Heap size: 100000 bytes

Limits: 4 to 2048 bytes

Allocation time: 0.060608s

free time: 0.000484s

Usage: 0.963329%

KMEM ALLOCATOR LOWER TEST:

Requests: 100000

Heap size: 100000 bytes

Limits: 4 to 2048 bytes

Allocation time: 0.013203s

free time: 0.000379s

Usage: 0.760518%

LIST ALLOCATOR MIDDLE TEST:

Requests: 100000

Heap size: 100000 bytes

Limits: 4 to 81920 bytes

Allocation time: 0.005259s

free time: 0.000360s

Usage: 0.996717%

KMEM ALLOCATOR MIDDLE TEST:

Requests: 100000

Heap size: 100000 bytes

Limits: 4 to 81920 bytes

Allocation time: 0.011377s

free time: 0.000454s

Usage: 0.937181%

LIST ALLOCATOR UPPER TEST:

Requests: 100000

Heap size: 100000 bytes

Limits: 4 to 4096000 bytes

Allocation time: 0.001643s

free time: 0.000408s

Usage: 0.996744%

KMEM ALLOCATOR UPPER TEST:

Requests: 100000

Heap size: 100000 bytes

Limits: 4 to 4096000 bytes

Allocation time: 0.011723s

free time: 0.000398s

Usage: 0.995156%

kirill@kirill:~/localProjects/OSlabs/kp/build/src$ /home/kirill/localProjects/OSlabs/kp/build/src/main 1000

LIST ALLOCATOR LOWER TEST:

Requests: 100000

Heap size: 1000 bytes

Limits: 4 to 2048 bytes

Allocation time: 0.008314s

free time: 0.000742s

Usage: 0.958457%

KMEM ALLOCATOR LOWER TEST:

Requests: 100000

Heap size: 1000 bytes

Limits: 4 to 2048 bytes

Allocation time: 0.005569s

free time: 0.000606s

Usage: 0.948486%

LIST ALLOCATOR MIDDLE TEST:

Requests: 100000

Heap size: 1000 bytes

Limits: 4 to 81920 bytes

Allocation time: 0.003154s

free time: 0.000618s

Usage: 0.916667%

KMEM ALLOCATOR MIDDLE TEST:

Requests: 100000

Heap size: 1000 bytes

Limits: 4 to 81920 bytes

Allocation time: 0.003092s

free time: 0.000664s

Usage: 0.789551%

LIST ALLOCATOR UPPER TEST:

Requests: 100000

Heap size: 1000 bytes

Limits: 4 to 4096000 bytes

Allocation time: 0.002459s

free time: 0.000831s

Usage: 0.942308%

KMEM ALLOCATOR UPPER TEST:

Requests: 100000

Heap size: 1000 bytes

Limits: 4 to 4096000 bytes

Allocation time: 0.003254s

free time: 0.000669s

Usage: 0.773438%

Сравнение алгоритмов аллокации

Изучая результаты работы программы не трудно сделать вывод, что алгоритм свободных блоков показал себя очень уверенно во всех типах тестов, учитывая простоту его реализации, это делает его еще более востребованным. Однако он обладает большим недостатком, связанным с фрагментацией памяти, что может выявиться при частых выделениях и освобождениях памяти.

Алгоритм Мак-Кьюзика-Кэрелса показал себе не очень хорошо, при работе с большими блоками данных (ввиду того, что они чаще всего занимают по несколько страниц), но для выделения небольших участков алгоритм прекрасно подходит, из-за разбиения страниц на заготовленные буферы.

Если же оценивать фактор использования алгоритмов, то не сложно предугадать победу алгоритма свободных блоков, ведь в алгоритме Мака-Кьюзика-Кэрелса буферы имеют фиксированный размер, кратный степени 2, поэтому он могут быть заполнены не до конца (худший случай $2^n + 1$ байт).

Вывод

В результате данной лабораторной работы, я приобрел важные практические навыки в использовании знаний, полученных в течении курса и провел исследование алгоритмов аллокации памяти. Я реализовал 2 алгоритма: Мака-Кьюзика-Кэрелса и списка свободных блоков (первое вхождение). В ходе выполнения работы было выяснено, что алгоритм свободных блоков показал себя уверенно, продемонстрировав высокий фактор использования и низкую скорость аллокации, к тому же он прост в реализации. Однако он имеет проблемы с фрагментацией памяти. Алгоритм Мак-Кьюзика-Кэрелса хорошо себя показывает на небольших данных, чей размер стремится к степеням двойки. Данное задание оказалось очень интересным, я смог узнать много нового про работу с памятью в ОС, а также в языке СИ. Также я приобрел понимание принципа работы аллокаторов, что пригодится в дальнейшей работе, ведь не существует универсального алгоритма аллокации памяти, ввиду особенностей данных или устройства ОС, поэтому необходимо уметь реализовывать свои аллокаторы.