Московский авиационный институт
(национальный исследовательский университет)
Институт № 8 «Информационные технологии и прикладная математика»

Курсовой проект
по курсу «Операционные системы»

Сравнение алгоритмов аллокации памяти

Студент: Минеева Светлана Алексеевна

Группа: М8О-210Б-21

Преподаватель: Миронов Е.С

Вариант №14
Оценка: ____
Дата: 26.12.2022
Подпись: _____

Москва, 2022

Содержание:

1. Цель курсового проекта

- Приобретение практических навыков в использовании знаний, полученных в течении курса

- Проведение исследования в выбранной предметной области

2. 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.

3. Вариант задания

Аллокаторы памяти

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

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

Каждый аллокатор памяти должен иметь функции аналогичные стандартным функциям free и malloc (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:

- Подробное описание каждого из исследуемых алгоритмов
- Процесс тестирования
- Обоснование подхода тестирования
- Результаты тестирования
- 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-Кьюзика-Кэрелса.

4. Аллокаторы памяти

Операционная система управляет всей доступной физической памятью машины и производит ее выделение для остальных подсистем ядра и прикладных задач. Данной процедурой управляет ядро, оно же и освобождает память, когда это требуется.

Менеджером памяти (аллокатором) называется часть ОС, непосредственно обрабатывающая запросы на выделение и освобождение памяти.

Существуют разные алгоритмы для реализации аллокаторов. Каждый из них имеет свои особенности и недостатки. Согласно моему варианту курсовой работы я реализовала следующие алгоритмы аллокации:

- Алгоритм с выбором первого подходящего участка памяти, основанный на списках;
- Алгоритм Мак-Кьюзика-Кэрелса.

Рассмотрим подробнее алгоритмы: их реализации и характеристики.

4.1. Списки свободных блоков (первое подходящее)

Этот способ отслеживает память с помощью связных списков распределенных и свободных сегментов памяти, где сегмент содержит либо свободную, либо выделенную память. Каждый элемент списка хранит внутри

свое обозначение — является ли он хранилищем выделенной или освобожденной памяти, а также размер участка памяти и указатель на его начало.

Список поддерживает инвариант отсортированности элементов по адресам с самой инициализации аллокатора. Благодаря этому, упрощается обновление списка при выделении или освобождении памяти. Для таких списков существует 4 алгоритма выделения памяти:

- Первое подходящее — список сегментов сканируется, пока не будет найдено пустое пространство подходящего размера. После этого сегмент разбивается на два сегмента, один из которых будет пустым. Данный алгоритм довольно быстр, ведь поиск ведется с наименьшими затратами времени.
- Следующее подходящее — работает примерно так же, как и предыдущий алгоритм, за исключением того, что запоминает свое местоположение при выделении. При следующем запросе на выделение памяти поиск начинается с того места, на котором алгоритм остановился в предыдущий раз. Исследование работы алгоритма показало, что его производительность несколько хуже, чем у «первого подходящего»
- Наиболее подходящее — при нем ведется линейный поиск наименьшего по размеру подходящего сегмента. Это делается для того, чтобы наилучшим образом соответствовать запросу и имеющимся пустым пространствам в списке
- Наименее подходящее — алгоритм, противоположный вышеописанному — при выделении используется наибольший возможный сегмент памяти. Моделирование показало, что использование данного алгоритма не является хорошей идеей.

Далее речь будет идти об алгоритме «первое подходящее». Этот алгоритм работает быстрее, чем «наиболее подходящее» (так как при каждом запросе не ведется поиск по всему списку). Его применение также приводит к менее расточительному использованию памяти, чем использование «наиболее подходящего» или «следующего подходящего».

Работа всех вышеописанных алгоритмов может быть ускорена за счет ведения отдельных списков для занятых и для пустых пространств. Это ускоряет выделение памяти, но замедляет процедуру освобождения памяти. Так или иначе, даже при всех улучшениях данные алгоритмы достаточно сильно страдают от фрагментации.

4.2. Алгоритм Мак-Кьюзика-Кэрелса

Маршалл Кирк Мак-Кьюзик и Майкл Дж. Кэрелс разработали усовершенствованный метод простых списков, основанных на степени двойки. 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. Каждая страница может находиться в одном из трех перечисленных состояний:

- Быть свободной. В этом случае дополнительный массив 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 дополнительного массива. Отсутствие заголовка в выделенных буферах позволяет экономить память при удовлетворении запросов с потребностью в памяти, кратной некоторой степени числа 2.

Данный 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.

5. Реализация аллокаторов

allocator_list.h

```
#ifndef ALLOCATOR_LIST_H
#define ALLOCATOR_LIST_H

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

typedef unsigned char* PBYTE_LIST;

typedef struct block_list {
  size_t size;
  struct block_list* previous;
  struct block_list* next;
} block_list;

static block_list* begin_list;
static block_list* free_list;
static size_t size_list;
static size_t request_list = 0;
static size_t total_list = 0;

int initialization_list(size_t size);
void destroy_list();
void* alloc_block_list(block_list* block, size_t size);
void* malloc_list(size_t size);
void list_free(void* address);
size_t get_request_list();
size_t get_total_list();

#endif
```

allocator_list.c

```
#include "allocator_list.h"

int initialization_list(size_t size) {
  if(size < sizeof(block_list)) {
    size = sizeof(block_list);
  }

  begin_list = (block_list*)malloc(size);

  if(begin_list == NULL) {
    return 0;
  }

  begin_list->size = size;
  begin_list->previous = NULL;
  begin_list->next = NULL;
  free_list = begin_list;
```

```
  size_list = size;

  return 1;
}

void destroy_list() {
  free(begin_list);
}

void* alloc_block_list(block_list* block, size_t size) {
  block_list* next_block = NULL;

  if(block->size >= size + sizeof(block_list)) {
    next_block = (block_list*)((PBYTE_LIST)block + size);
    next_block->size = block->size - size;
    next_block->previous = block->previous;
    next_block->next = block->next;
    block->size = size;

    if(block->previous != NULL) {
      block->previous->next = next_block;
    }

    if(block->next != NULL) {
      block->next->previous = next_block;
    }

    if(block == free_list) {
      free_list = next_block;
    }
  }
  else {
    if(block->previous != NULL) {
      block->previous->next = block->next;
    }

    if(block->next != NULL) {
      block->next->previous = block->previous;
    }

    if(block == free_list) {
      free_list = block->next;
    }
  }

  return (void*)((PBYTE_LIST)block + sizeof(size_t));
}

void* malloc_list(size_t size) {
  size_t first_size = size_list;
  size_t old_size = size;
  block_list* first_block = free_list;
  block_list* current = free_list;
```

```
  size += sizeof(size_t);

  if(size < sizeof(block_list)) {
    size = sizeof(block_list);
  }

  int flag = 0;

  while(current != NULL && flag==0) {
    if (current->size < first_size && current->size >= size) {
      first_size = current->size;
      first_block = current;
      flag = 1;
    }

    current = current->next;
  }

  if(free_list == NULL || first_block->size < size) {
    return NULL;
  }

  request_list += old_size;
  total_list += size;
  return alloc_block_list(first_block, size);
}

void list_free(void* address) {
  block_list* block = (block_list*)((PBYTE_LIST)address - sizeof(size_t));
  block_list* current = free_list;
  block_list* left_block = NULL;
  block_list* right_block = NULL;

  while(current != NULL) {
    if((block_list*)((PBYTE_LIST)current + current->size) <= block) {
      left_block = current;
    }

    if((block_list*)((PBYTE_LIST)block + block->size) <= current) {
      right_block = current;
      break;
    }

    current = current->next;
  }

  if(left_block != NULL) {
    left_block->next = block;
  }
  else {
    free_list = block;
  }
```

```
  if(right_block != NULL) {
    right_block->previous = block;
  }

  block->previous = left_block;
  block->next = right_block;
  current = free_list;

  while(current != NULL) {
    if ((block_list*)((PBYTE_LIST)current + current->size) == current->next) {
      current->size += current->next->size;
      current->next = current->next->next;

      if (current->next != NULL) {
        current->next->previous = current;
      }

      continue;
    }

    current = current->next;
  }
}

size_t get_request_list() {
  return request_list;
}

size_t get_total_list() {
  return total_list;
}
```

allocator_mkk.h

```
#include <stdlib.h>

typedef unsigned char* PBYTE_MKK;

typedef enum memory_structure {
  free_state = 0
} memory_state;

typedef struct block_mkk_structure {
  struct block_mkk_structure* next;
} block_mkk;

static const size_t PAGE_SIZE_MKK = 4096;
static void* heap_mkk = NULL;
static size_t* memory_size_mkk = NULL;
static block_mkk** list_mkk = NULL;
static size_t pages_mkk = 0;
```

```
static size_t pow_mkk = 0;
static size_t pow_index_minimum = 0;
static size_t request_mkk = 0;
static size_t total_mkk = 0;

int initialization_mkk(size_t size);
void destroy_mkk();
void* malloc_mkk(size_t size);
void free_mkk(void* address);
block_mkk* alloc_page_mkk(size_t size);
void free_page_mkk(block_mkk* block);
void split_page_mkk(block_mkk* block, size_t powIndex);
size_t pow_of_size_mkk(size_t size);
size_t get_pages_count_mkk(size_t size);
size_t get_page_index_mkk(block_mkk* block);
size_t get_request_mkk();
size_t get_total_mkk();

#endif
```

allocator_mkk.c

```
#include "allocator_mkk.h"

int initialization_mkk(size_t size) {
  size_t index;
  block_mkk* block = NULL;

  pages_mkk = get_pages_count_mkk(size);
  pow_mkk = pow_of_size_mkk(PAGE_SIZE_MKK);
  pow_index_minimum = pow_of_size_mkk(sizeof(block_mkk));
  heap_mkk = malloc(pages_mkk * PAGE_SIZE_MKK);
  memory_size_mkk = (size_t*)malloc(sizeof(size_t) * pages_mkk);
  list_mkk = (block_mkk**)malloc(sizeof(block_mkk*) * pow_mkk);

  if(heap_mkk == NULL || memory_size_mkk == NULL || list_mkk == NULL) {
    return 0;
  }

  memory_size_mkk[free_state] = free_state;
  list_mkk[free_state] = (block_mkk*)heap_mkk;
  block = list_mkk[free_state];

  for(index = 1; index < pages_mkk; ++index) {
    memory_size_mkk[index] = free_state;
    block->next = (block_mkk*)((PBYTE_MKK)block + PAGE_SIZE_MKK);
    block = block->next;
  }

  block->next = NULL;

  for(index = 1; index < pow_mkk; ++index) {
```

```
    list_mkk[index] = NULL;
  }

  return 1;
}

void destroy_mkk() {
  free(heap_mkk);
  free(memory_size_mkk);
  free(list_mkk);
}

void* malloc_mkk(size_t size) {
  size_t pow_index = pow_of_size_mkk(size);
  size_t old_size = size;
  block_mkk* block = NULL;

  if(pow_index < pow_index_minimum) {
    pow_index = pow_index_minimum;
  }

  size = 1 << pow_index;

  if(size < PAGE_SIZE_MKK) {
    if(list_mkk[pow_index] == NULL) {
      block = alloc_page_mkk(size);

      if(block == NULL) {
        return NULL;
      }

    split_page_mkk(block, pow_index);
    }

    block = list_mkk[pow_index];
    list_mkk[pow_index] = block->next;

    request_mkk += old_size;
    total_mkk += size;

    return (void*)block;
  }
  else {
    request_mkk += old_size;
    total_mkk += size;

    return alloc_page_mkk(size);
  }
}

void free_mkk(void* address) {
  size_t page_index = get_page_index_mkk((block_mkk*)address);
  size_t pow_index = pow_of_size_mkk(memory_size_mkk[page_index]);
```

```
  block_mkk* block = (block_mkk*)address;

  if(memory_size_mkk[page_index] < PAGE_SIZE_MKK) {
    block->next = list_mkk[pow_index];
    list_mkk[pow_index] = block;
  }
  else {
    free_page_mkk(block);
  }
}

block_mkk* alloc_page_mkk(size_t size) {
  size_t count = 0;
  size_t page_index = 0;
  size_t previous_index = get_page_index_mkk(list_mkk[free_state]);
  size_t pages = get_pages_count_mkk(size);
  block_mkk* current = list_mkk[free_state];
  block_mkk* previous = NULL;
  block_mkk* page = NULL;

  while(current != NULL) {
    page_index = get_page_index_mkk(current);

    if(page_index - previous_index <= 1) {
      if(page == NULL) {
        page = current;
      }

      ++count;
    }
    else {
      page = current;
      count = 1;
    }

    if(count == pages) {
      break;
    }

    previous = current;
    current = current->next;
    previous_index = page_index;
  }

  if(count < pages) {
    page = NULL;
  }

  if(page != NULL) {
    page_index = get_page_index_mkk(page);
    memory_size_mkk[page_index] = size;
    current = (block_mkk*)((PBYTE_MKK)page + (pages - 1) * PAGE_SIZE_MKK);
```

```
    if (previous != NULL) {
      previous->next = current->next;
    }
    else {
      list_mkk[free_state] = current->next;
    }
  }

  return page;
}

void free_page_mkk(block_mkk* block) {
  size_t index;
  size_t page_index = get_page_index_mkk(block);
  size_t block_count = memory_size_mkk[page_index] / PAGE_SIZE_MKK;
  block_mkk* left = NULL;
  block_mkk* right = NULL;
  block_mkk* current = block;

  while(current != NULL) {
    if (current < block) {
      left = current;
    }
    else {
      if(current > block) {
        right = current;

        break;
      }
    }

    current = current->next;
  }

  for(index = 1; index < block_count; ++index) {
    block->next = (block_mkk*)((PBYTE_MKK)block + PAGE_SIZE_MKK);
    block = block->next;
  }

  block->next = right;

  if(left != NULL) {
    left->next = block;
  }
  else {
    list_mkk[free_state] = block;
  }
}

void split_page_mkk(block_mkk* block, size_t pow_index) {
  size_t index;
  size_t page_index = get_page_index_mkk(block);
  size_t block_size = 1 << pow_index;
```

```
  size_t block_count = PAGE_SIZE_MKK / block_size;

  list_mkk[pow_index] = block;
  memory_size_mkk[page_index] = block_size;

  for(index = 1; index < block_count; ++index) {
    block->next = (block_mkk*)((PBYTE_MKK)block + block_size);
    block = block->next;
  }

  block->next = NULL;
}

size_t pow_of_size_mkk(size_t size) {
  size_t pow = 0;

  while(size > ((size_t)1 << pow)) {
    ++pow;
  }

  return pow;
}

size_t get_pages_count_mkk(size_t size) {
  return size / PAGE_SIZE_MKK + (size_t)(size % PAGE_SIZE_MKK != 0);
}

size_t get_page_index_mkk(block_mkk* block) {
  return (size_t)((PBYTE_MKK)block - (PBYTE_MKK)heap_mkk) / PAGE_SIZE_MKK;
}

size_t get_request_mkk() {
  return request_mkk;
}

size_t get_total_mkk() {
  return total_mkk;
}
```

main.c

```
#include <stdio.h>
#include <time.h>
#include "allocator_list.h"
#include "allocator_mkk.h"

typedef struct request_structure {
  void* address;
  size_t bytes;
} request;

size_t parse_size(const char* string) {
```

```
  size_t size = 0;

  while(*string != '\0') {
    if(*string < '0' || *string > '9') {
      return 0;
    }

    size = size * 10 + *string - '0';
    ++string;
  }

  return size;
}

int main(int argument_count, char* argument_vector[]) {
  const size_t NUMBER_REQUESTS = 1000;
  const size_t MAX_BYTES = 5000;
  clock_t first_time;
  clock_t second_time;
  size_t first_index;
  size_t second_index;
  size_t third_index;
  size_t argument;
  size_t query = 0;
  size_t total = 0;
  size_t* permute = (size_t*)malloc(sizeof(size_t) * NUMBER_REQUESTS);
  request* requests = (request*)malloc(sizeof(request) * NUMBER_REQUESTS);

  srand((unsigned int)time(0));

  if(argument_count < 2) {
    printf("Usage: %s <SIZE>\n", argument_vector[0]);

    return 0;
  }

  argument = parse_size(argument_vector[1]);

  if(!initialization_list(argument)) {
    printf("Error. No memory\n");

    return 0;
  }

  if(!initialization_mkk(argument)) {
    printf("Error. No memory\n");

    return 0;
  }

  for(first_index = 0; first_index < NUMBER_REQUESTS; ++first_index) {
    requests[first_index].bytes = 1 + rand() % MAX_BYTES;
    permute[first_index] = first_index;
```

```
}

for(first_index = 0; first_index < NUMBER_REQUESTS; ++first_index) {
  second_index = rand() % NUMBER_REQUESTS;
  third_index = rand() % NUMBER_REQUESTS;
  argument = permute[second_index];
  permute[second_index] = permute[third_index];
  permute[third_index] = argument;
}

printf("Alloc requests: %zu\n", NUMBER_REQUESTS);
printf("Bytes: 1 to %zu\n\n", MAX_BYTES);
printf("Allocator LIST:\n");

first_time = clock();

for(first_index = 0; first_index < NUMBER_REQUESTS; ++first_index) {
  requests[first_index].address = malloc_list(requests[first_index].bytes);
}

second_time = clock();

printf("Alloc time: %lf\n", (double)(second_time - first_time) / CLOCKS_PER_SEC);

query = get_request_list();
total = get_total_list();

for(first_index = 0; first_index < NUMBER_REQUESTS; ++first_index) {
  if(requests[permute[first_index]].address == NULL) {
    continue;
  }

  list_free(requests[permute[first_index]].address);
}

first_time = clock();

printf("Free time: %lf\n", (double)(first_time - second_time) / CLOCKS_PER_SEC);
printf("Usage factor: %lf\n\n", (double)query / total);
printf("Allocator MKK:\n");

first_time = clock();

for(first_index = 0; first_index < NUMBER_REQUESTS; ++first_index) {
  requests[first_index].address = malloc_mkk(requests[first_index].bytes);
}

second_time = clock();

printf("Alloc time: %lf\n", (double)(second_time - first_time) / CLOCKS_PER_SEC);

query = get_request_mkk();
total = get_total_mkk();
```

```
  for(first_index = 0; first_index < NUMBER_REQUESTS; ++first_index) {
    if(requests[permute[first_index]].address == NULL) {
      continue;
    }

    free_mkk(requests[permute[first_index]].address);
  }

  first_time = clock();

  printf("Free time: %lf\n", (double)(first_time - second_time) / CLOCKS_PER_SEC);
  printf("Usage factor: %lf\n", (double)query / total);

  destroy_list();
  destroy_mkk();

  free(requests);
  free(permute);

  return 0;
}
```

6. Демонстрация работы программы

```
Last login: Mon Dec 26 12:53:07 on ttys000

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
MacBook-Pro-MacBook:~ macbookpro$ gcc allocator_list.c allocator_mkk.c
main.c -o allocators
MacBook-Pro-MacBook:~ macbookpro$ ./allocators 8000
Alloc requests: 1000
Bytes: 1 to 5000

Allocator LIST:
Alloc time: 0.000009
Free time: 0.000010
Usage factor: 0.989999

Allocator MKK:
Alloc time: 0.000041
Free time: 0.000006
Usage factor: 0.655958
MacBook-Pro-MacBook:~ macbookpro$ ./allocators 4097
Alloc requests: 1000
Bytes: 1 to 5000

Allocator LIST:
Alloc time: 0.000026
Free time: 0.000018
Usage factor: 0.984364
```

```
Allocator MKK:
Alloc time: 0.000066
Free time: 0.000007
Usage factor: 0.658396
MacBook-Pro-MacBook:~ macbookpro$ ./allocators 1024
Alloc requests: 1000
Bytes: 1 to 5000

Allocator LIST:
Alloc time: 0.000012
Free time: 0.000011
Usage factor: 0.968096

Allocator MKK:
Alloc time: 0.000050
Free time: 0.000006
Usage factor: 0.665707
MacBook-Pro-MacBook:~ macbookpro$
```

7. Сравнение алгоритмов аллокации

Исходя из предыдущего пункта - демонстрации работы программы, нетрудно сравнить два алгоритма аллокации. Само тестирование проводилось на 1000 запросах по выделению и освобождению блоков размером от 1 до 5000 бит, размер каждого следующего запроса выбирается рандомно с учётом заданного диапазона. Изначальное количество свободных бит для данной работы задается с помощью параметра при запуске программы.

По скорости выделения блоков алгоритм свободных блоков (первое подходящее) быстрее в 3-4 раза. Данный результат легко объяснить: в этом алгоритме мы используем первый подходящий по размеру блок, из-за чего не требуется проход по всему списку блоков, что здорово увеличивает скорость выделения блоков.

По скорости освобождения блоков первый и последний раз лидером становится алгоритм Мак-Кьюзика-Кэрелса, однако разница скоростей алгоритмов не слишком-то и велика. Данная заслуга отдается страничному представлению памяти и хранению указателей на каждый занятый блок.

Сравнивая алгоритмы по фактору использования, фактор использования – это соотношение общего объёма запрашиваемой памяти к объёму, позволяющему удовлетворить эти запросы, можно увидеть явную победу алгоритма свободных блоков. В алгоритме Мак-Кьюзика-Кэрелса страницы памяти задаются одинаковым для всех числом степени двойки, что часто приводит к созданию нетребуемой памяти, из-за этого и стабильное снижение фактора использования.

Алгоритм свободных блоков (первое подходящее) гораздо легче понять и реализовать, что также немало важно. В итоге, алгоритм свободных блоков (первое подходящее) выигрывает практически по всем параметрам сравнения.

8. Вывод:

В процессе выполнения курсовой работы я приобрела важные практические навыки в использовании знаний, полученных в течении курса и провела исследование в области аллокации памяти. Я реализовала и сравнила два алгоритма аллокации памяти: алгоритм Мак-Кьюзика-Кэрелса и алгоритм свободных списков (первое подходящее). Из сравнения стало ясно, что алгоритм Мак-Кьюзика-Кэрелса обходит алгоритм свободных списков только по скорости освобождения блоков, в остальном же, а именно в скорости освобождения блоков и факторе использования, алгоритм свободных блоков выигрывает. Да и в целом, реализация и сама суть алгоритма свободных блоков намного проще к пониманию. Данное интересное задание может пригодиться мне в дальнейшем будущем, поскольку иногда встроенные аллокаторы памяти могут не удовлетворять требуемой к выполнению задаче, в таком случае нужно создавать аллокатор памяти самому.