Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Курсовой проект по курсу

«Операционные системы»

Тема работы

«Аллокаторы памяти»

Студент: Старцев Иван Романовмч
Группа: М8О-201Б-21
Вариант: 20
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2022

Содержание

Репозиторий

https://github.com/IvanTvardovsky/OS-labs

Постановка задачи

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. 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

Вариант 20:

Необходимо сравнить два алгоритма аллокации: списки свободных блоков (наиболее подходящее) и алгоритм Мак-Кьюзи-Кэрлса

Подробное описание каждого из исследуемых алгоритмов

Списки 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 её общему объёму памяти. После этого клиенты начинают запрашивать и освобождать участки памяти, вследствие чего область становится фрагментированной. Ядро создаёт для каждого нового последовательного свободного участка памяти новое вхождение карты. Элементы карты сортируются в порядке возрастания адресов, что упрощает

задачу слияния свободных участков.
Наиболее подходящий участок. Выделение памяти из наиболее подходящей свободной области, имеющей достаточный для удовлетворения запроса объём. Это самый выгодный по памяти алгоритм из всех трёх (первый подходящий участок, наиболее походящий участок, наименее подходящий участок), но он не самый быстрый.

Алгоритм Мак-Кьюзи-Кэрелса:

Маршалл Кирк Мак-Кьюзик и Майкл Дж. Кэрлс разработали усовершенствованный метол выделения памяти, который был реализован во многих вариантах системы UNIX. Методика позволяет избавиться от потерь в тех случаях, когда размер запрашиваемого участка памяти равен некоторой степени двойки. В нём 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).
Каждая страница может находиться в одном из трёх перечисленных состояний.

- Быть свободной.
- Быть разбитой на буферы определённого размера.
- Являться частью буфера, объединяющего сразу несколько страниц.

Вызов процедуры malloc() заменён макроопределением, которое производит округления значения длины запрашиваемого участка вверх до достижения числа, являющегося степенью двойки (при этом не нужно прибавлять какие-либо дополнительные байты на заголовок) и удаляет буфер из соответствующего списка свободных буферов. Макрос вызывает функцию malloc() для запроса одной или нескольких страниц тогда, когда список свободных буферов необходимого размера пуст. В этом случае malloc() вызывает процедуру, которая берёт свободную страницу и разделяет её на буферы необходимого размера. Здесь цикл заменён на схему вычислений по условию.

Процесс тестирования и обоснование процесса тестирования:

В запросы: requests, состоящие из адресов и размеров запросов псевдослучайным образом вмещаются значения от 1 до MAX_BYTES, причём счетом запросов является NUMBER_REQUESTS. еще создаётся массив permute из NUMBER_REQUESTS индексов запросов requests, причём эти индексы псевдослучайным образом перемешиваются. Потом мы пробуем

выделить место для каждого запроса и как раз освобождаем запросы псевдослучайным образом. Таким образом, при тестировании сведены к минимуму потери точности из-за накладных расходов при измерении ключевых характеристик: скорости выделения, освобождения памяти и фактора использования.

Исходный код

main.cpp

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

using namespace std;

// запросы содержат адрес и размер запрашиваемого места
typedef struct request_struce {
void* address;
size_t bytes;
} request;

// char* в число
size_t ToSizeT(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 REQUEST_QUANTITY = 1000;
const size_t MAX_BYTES = 5000;
clock_t first_time, second_time;
size_t first_index, second_index, third_index;
size_t argument;
size_t query = 0;
size_t total = 0;
size_t* permute = (size_t*)malloc(sizeof(size_t) * REQUEST_QUANTITY);
request* requests = (request*)malloc(sizeof(request) * REQUEST_QUANTITY);

// генератор случайных чисел
srand((unsigned int)time(0));

argument = ToSizeT(argument_vector[1]);
```

```
// аллокация списка свободных блоков
if (!InitializationList(argument)) {
cout << "Error. No memory\n";
return 0;
}
// аллокация Мак-Кьюзи-Кэрлса
if (!MKKInitialization(argument)) {
cout << "Error. No memory\n";
return 0;
}

// псевдослучайные запросы и массив для них
for (first_index = 0; first_index < REQUEST_QUANTITY; ++first_index) {
requests[first_index].bytes = 1 + rand() % MAX_BYTES;
permute[first_index] = first_index;
}

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

cout << "Количество запросов: " << REQUEST_QUANTITY << "\n";
cout << "Байт: " << MAX_BYTES << "\n\n";
cout << "Аллокация списка свободных блоков:\n";

first_time = clock();

// для каждого запроса ищем свободный блок
for (first_index = 0; first_index < REQUEST_QUANTITY; ++first_index) {
requests[first_index].address = malloc_list(requests[first_index].bytes);
}
second_time = clock();

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

// снова счетчики
query = ListOfRequests();
total = TotalList();

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

printf("Очистка заняла: %lf\n", (double)(first_time - second_time) /
CLOCKS_PER_SEC);
cout << "КПД использованной и запрошенной памяти: " << (long double)query /
total << "\n\n";
cout << "Аллокация Мак-Кьюзи-Кэрлса\n";
```

```
first_time = clock();
for (first_index = 0; first_index < REQUEST_QUANTITY; ++first_index) {
requests[first_index].address = MKKMalloc(requests[first_index].bytes);
}
second_time = clock();

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

// снова счетчики
query = MKKRequest();
total = MKKTotal();

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

printf("Очистка заняла: %lf\n", (double)(first_time - second_time) /
CLOCKS_PER_SEC);
cout << "КПД использования и запрошенной памяти: " << (long double)query /
total << "\n";

Destroy();
MKKDestroy();

free(requests);
free(permute);

return 0;
}
```

allocator_list.h

```
#ifndef ALLOCATOR_LIST_H
#define ALLOCATOR_LIST_H

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

typedef unsigned char* PBYTE_LIST;

// Структура двусвязного списка для свободных блоков
typedef struct block_list {
size_t size;
struct block_list* previous;
struct block_list* next;
} block_list;

//Стартовый адрес области
static block_list* begin_list;

//Адрес первого свободного блока
```

```
static block_list* free_list;

//Общий размер выделенной памяти
static size_t size_list;

//Счётчик количества запрашиваемой информации
static size_t request_list = 0;

//Счётчик количества используемой информации
static size_t total_list = 0;

int InitializationList(size_t size);
void Destroy();
void* BlockAlloc(block_list* block, size_t size);
void* malloc_list(size_t size);
void FreeList(void* address);
size_t ListOfRequests();
size_t TotalList();

#endif
```

allocator_mkk.h

```
#ifndef ALLOCATOR_MKK_H
#define ALLOCATOR_MKK_H

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

typedef unsigned char* PBYTE_MKK;

typedef enum memory_structure {
free_state = 0
} memory_state;

// каждая из страниц может находиться в одном из 3 состояний:
// 1. свободная ->
// -> соответствующий элемент массива содержит указатель на элемент, описывающий следующую свободную страницу
// 2. разбитая на буферы определённого размера (некоторая степень 2) -> элемент массива содержит размер буфера
// 3. частью буфера, объединяющего сразу несколько страниц ->
// -> элемент массива указывает на первую страницу буфера, в котором находятся данные о его длине

// односвязный список для блоков
typedef struct block_mkk_structure {
struct block_mkk_structure* next;
} block_mkk;

// размер одной страницы
static const size_t PAGE_SIZE_MKK = 4096;

// стартовый адрес области
static void* heap_mkk = NULL;

// массив для управления страницами
```

```
static size_t* memory_size_mkk = NULL;

//Массив, содержащий заголовки всех буферов, имеющих размер меньше одной страницы
static block_mkk** list_mkk = NULL;

//Общее количество страниц
static size_t pages_mkk = 0;

// Степень двойки
static size_t pow_mkk = 0;

//Минимальный размер необходимый для хранения указателя на элемент
static size_t pow_index_minimum = 0;

//Счётчик количества запрашиваемой информации
static size_t request_mkk = 0;

//Счётчик количества использованной информации
static size_t total_mkk = 0;

int MKKInitialization(size_t size);
void MKKDestroy();
void* MKKMalloc(size_t size);
void MKKFree(void* address);
block_mkk* MKKAllocPage(size_t size);
void MKKFreePage(block_mkk* block);
void MKKSplitPage(block_mkk* block, size_t powIndex);
size_t PowOfTwo(size_t size);
size_t MKKPageCounter(size_t size);
size_t MKKPageIndex(block_mkk* block);
size_t MKKRequest();
size_t MKKTotal();

#endif
```

allocator_list.cpp

```
#include "../include/allocator_list.h"

int InitializationList(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() {
free(begin_list);
}

void* BlockAlloc(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) {
// если блок может вместить необходимую информацию, значит он подходит
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;

// добавляем в счетчики
request_list += old_size;
total_list += size;

// фрагментация найденного блока для использования оставшего места
return BlockAlloc(first_block, size);
}

// освобождение места занимаемого запросом по адресу address
void FreeList(void* address) {
// находим адрес блока
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) {
// блок, который располагается левее освобождённого и самый близкий
if ((block_list*)((PBYTE_LIST)current + current->size) <= block) left_block = current;

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

current = current->next;
}
// добавление освобождённого блока в двусвязный список свободных блоков
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;

// объединение двух блоков рядом в один блок большего размера
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;
}
}

// счетчики
size_t ListOfRequests() {
return request_list;
```

```
}

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

allocator_mkk.cpp

```
#include "../include/allocator_mkk.h"

int MKKInitialization(size_t size) {
size_t index;
block_mkk* block = NULL;
// память разбита на набор последовательных страниц
pages_mkk = MKKPageCounter(size);
// степень двойки для размера одной страницы
pow_mkk = PowOfTwo(PAGE_SIZE_MKK);
// степень двойки для структуры блока
pow_index_minimum = PowOfTwo(sizeof(block_mkk));
// стартовый адрес области
heap_mkk = malloc(pages_mkk * PAGE_SIZE_MKK);
// массив управления страницами
memory_size_mkk = (size_t*)malloc(sizeof(size_t) * pages_mkk);
// заголовки буферов размер которых меньше одной страницы для
объединения
list_mkk = (block_mkk**)malloc(sizeof(block_mkk*) * pow_mkk);

if (heap_mkk == NULL || memory_size_mkk == NULL || list_mkk == NULL) return 0;
// определение первой страницы и остальных
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;
// заголовков пока нет
for (index = 1; index < pow_mkk; ++index) {
list_mkk[index] = NULL;
}

return 1;
}

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

void* MKKMalloc(size_t size) {
size_t pow_index = PowOfTwo(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;
// если размер меньше размера страницы
if (size < PAGE_SIZE_MKK) {
if(list_mkk[pow_index] == NULL) {
// если нет ни одного буфера размером в 2^pow_index, то выделим страницу
для этого
block = MKKAllocPage(size);

if (block == NULL) return NULL;

//Разделим страницу которую мы выделили под буфер размером 2^pow_index
// на максимальное количество буферов размером 2^pow_index
MKKSplitPage(block, pow_index);
}
// первый свободный буфер
block = list_mkk[pow_index];
list_mkk[pow_index] = block->next;

// снова счётчики
request_mkk += old_size;
total_mkk += size;

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

return MKKAllocPage(size);
}
}

void MKKFree(void* address) {
size_t page_index = MKKPageIndex((block_mkk*)address);
size_t pow_index = PowOfTwo(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 MKKFreePage(block);
}

block_mkk* MKKAllocPage(size_t size) {
size_t count = 0;
size_t page_index = 0;
// находим где есть свободное место
size_t previous_index = MKKPageIndex(list_mkk[free_state]);
size_t pages = MKKPageCounter(size);
block_mkk* current = list_mkk[free_state];
block_mkk* previous = NULL;
block_mkk* page = NULL;
```

```
while (current != NULL) {
// определяем номер страницы, на котором находится буффер current
page_index = MKKPageIndex(current);

if (page_index - previous_index <= 1) {
if (page == NULL) page = current;
++count;
} else {
// если current занимает больше 1 страницы, то мы сбрасываем счётчик до 1,
// а в страницу добавляем current, чтобы при повторном прохождении цикла
не зайти в (if(page == NULL))
page = current;
count = 1;
}

if (count == pages) break;
// берем следующий буфер и обрабатываем текущий
previous = current;
current = current->next;
previous_index = page_index;
}

// не вместится на одну страницу
if (count < pages) page = NULL;

if (page != NULL) {
page_index = MKKPageIndex(page);
// страница page_index уже разделена на буферы размером с size
memory_size_mkk[page_index] = size;
// адрес текущего блока
current = (block_mkk*)((PBYTE_MKK)page + (pages - 1) * PAGE_SIZE_MKK);
if (previous != NULL) previous->next = current->next;
// вносим адрес свободного блока
else list_mkk[free_state] = current->next;
}

return page;
}

void MKKFreePage(block_mkk* block) {
size_t index;
size_t page_index = MKKPageIndex(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) {
// самый близкий левый буффер
if (current < block) left = current;
// аналогично правый
else {
if (current > block) {
right = current;
break;
```

```
}
}

current = current->next;
}

for (index = 1; index < block_count; ++index) {
// отделяем целые страницы от буфера
block->next = (block_mkk*)((PBYTE_MKK)block + PAGE_SIZE_MKK);
block = block->next;
}

block->next = right;

// освобождаем целые страницы
if (left != NULL) left->next = block;
else list_mkk[free_state] = block;
}

void MKKSplitPage(block_mkk* block, size_t pow_index) {
size_t index;
size_t page_index = MKKPageIndex(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;
// определение связи буферов на странице
for (index = 1; index < block_count; ++index) {
block->next = (block_mkk*)((PBYTE_MKK)block + block_size);
block = block->next;
}

block->next = NULL;
}

// степень двойки
size_t PowOfTwo(size_t size) {
size_t pow = 0;

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

return pow;
}

// необходимое количество страниц
size_t MKKPageCounter(size_t size) {
return size / PAGE_SIZE_MKK + (size_t)(size % PAGE_SIZE_MKK != 0);
}

// определение номера страницы, где block
size_t MKKPageIndex(block_mkk* block) {
return (size_t)((PBYTE_MKK)block - (PBYTE_MKK)heap_mkk) / PAGE_SIZE_MKK;
}
```

```
// снова счетчики
size_t MKKRequest() {
return request_mkk;
}

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

Результаты тестирования

```
tvard@tvard-HVY-WXX9:~/os/OS-labs/kp$ ./main 10
Количество запросов: 1000
Байт: 5000

Аллокация списка свободных блоков:
Заняло времени: 0.000012
Очистка заняла: 0.000027
КПД использованной и запрошенной памяти: 0.541667

Аллокация Мак-Кьюзи-Кэрлса
Заняло времени: 0.000053
Очистка заняла: 0.000009
КПД использования и запрошенной памяти: 0.650623
tvard@tvard-HVY-WXX9:~/os/OS-labs/kp$ ./main 10
Количество запросов: 1000
Байт: 5000

Аллокация списка свободных блоков:
Заняло времени: 0.000013
Очистка заняла: 0.000022
КПД использованной и запрошенной памяти: 0.541667

Аллокация Мак-Кьюзи-Кэрлса
Заняло времени: 0.000053
Очистка заняла: 0.000008
КПД использования и запрошенной памяти: 0.656274
tvard@tvard-HVY-WXX9:~/os/OS-labs/kp$ ./main 100
Количество запросов: 1000
Байт: 5000

Аллокация списка свободных блоков:
Заняло времени: 0.000012
Очистка заняла: 0.000024
КПД использованной и запрошенной памяти: 0.7

Аллокация Мак-Кьюзи-Кэрлса
Заняло времени: 0.000059
Очистка заняла: 0.000010
КПД использования и запрошенной памяти: 0.656411
tvard@tvard-HVY-WXX9:~/os/OS-labs/kp$ ./main 100
Количество запросов: 1000
Байт: 5000
```

Аллокация списка свободных блоков:
Заняло времени: 0.000012
Очистка заняла: 0.000024
КПД использованной и запрошенной памяти: 0.797468

Аллокация Мак-Кьюзи-Кэрлса
Заняло времени: 0.000060
Очистка заняла: 0.000009
КПД использования и запрошенной памяти: 0.663266
tvard@tvard-HVY-WXX9:~/os/OS-labs/kp$ ./main 1000
Количество запросов: 1000
Байт: 5000

Аллокация списка свободных блоков:
Заняло времени: 0.000012
Очистка заняла: 0.000026
КПД использованной и запрошенной памяти: 0.956434

Аллокация Мак-Кьюзи-Кэрлса
Заняло времени: 0.000054
Очистка заняла: 0.000009
КПД использования и запрошенной памяти: 0.656072
tvard@tvard-HVY-WXX9:~/os/OS-labs/kp$ ./main 1000
Количество запросов: 1000
Байт: 5000

Аллокация списка свободных блоков:
Заняло времени: 0.000013
Очистка заняла: 0.000019
КПД использованной и запрошенной памяти: 0.95988

Аллокация Мак-Кьюзи-Кэрлса
Заняло времени: 0.000054
Очистка заняла: 0.000009
КПД использования и запрошенной памяти: 0.654697
tvard@tvard-HVY-WXX9:~/os/OS-labs/kp$ ./main 1000
Количество запросов: 1000
Байт: 5000

Аллокация списка свободных блоков:
Заняло времени: 0.000004
Очистка заняла: 0.000009
КПД использованной и запрошенной памяти: 0.983903

Аллокация Мак-Кьюзи-Кэрлса
Заняло времени: 0.000021
Очистка заняла: 0.000003
КПД использования и запрошенной памяти: 0.653337

Заключение по проведённой работе

В ходе данного курсового проекта я приобрёл практические навыки в использовании знаний, полученных в течении курса, а также провёл исследование 2 аллокаторов памяти: список свободных блоков (наиболее подходящее) и алгоритм Мак-Кьюзи-Кэрлса.