Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

Курсовой проект по курсу

«Операционные системы»

Группа: М80-206Б-22

Студент: Жаднов М. Д.

Преподаватель: Миронов Е.С.

Оценка: _____

Дата: __.02.24

Москва, 2024

Постановка задачи

Цель работы

Сравнить 2 алгоритма аллокации: блоки по 2 в степени n и Мак-Кьюзика-Кэрелса.

Общий метод и алгоритм решения

Алгоритм блоков по 2 в степени n

Методика использует набор списков свободной памяти. В каждом списке хранятся буферы определенного размера. Размер буфера всегда кратен степени числа 2.

Каждый буфер имеет заголовок длиной в одно слово, этим фактом ограничивая возможности использования соотносимой с ним области памяти. Если буфер свободен, в его заголовке хранится указатель на следующий свободный буфер. В другом случае в заголовок буфера помещается указатель на список, в который он должен быть возвращен при освобождении. В некоторых реализациях заголовок содержит вместо этой информации размер выделенной области. Это позволяет обнаружить некоторые «баги», однако требует от процедуры free() вычисления местонахождения списка исходя из данных о размерах буферов.

Для выделения памяти клиент вызывает malloc(). В качестве входного аргумента функции передается желаемая величина участка. Распределитель вычисляет минимальный размер буфера, подходящий для удовлетворения запроса. Для этого необходимо прибавить к заданной величине слово, в котором будет размещен заголовок, и округлить полученное значение кверху до числа, являющегося степенью двойки. После вычисления распределитель извлекает буфер из соответствующего списка и оставляет в заголовке указатель на список свободных участков памяти. Процесс, запрашивающий участок памяти, получает в ответ указатель на следующий после заголовка байт.

Если клиент желает освободить буфер, то он вызывает для этой цели процедуру free(), входным аргументом которой является указатель, возвращенный malloc(). При этом нет необходимости уточнять размер буфера. Очевидно, что результатом выполнения операции #tee() станет освобождение буфера целиком. Технология не поддерживает возможности частичного освобождения участка памяти. Процедура free() производит обращение к заголовку буфера, откуда извлекает указатель на список свободных буферов и затем переносит его в этот список.

Распределитель памяти может быть проинициализирован путем предварительного выделения ему определенного количества буферов в каждом списке. Другой вариант предполагает создание изначально свободного списка, для заполнения которого вызывается распределитель страничного уровня.

Если список становится пустым, обработка последующего запроса malloc() для выделения участка памяти определенного размера может быть произведена одним из перечисленных способов:

- запрос блокируется до освобождения буфера подходящей длины;
- запрос удовлетворяется путем выделения буфера большего размера. Поиск свободного буфера начинается со следующего списка (по возрастанию размеров буферов) и продолжается до тех пор, пока не будет обнаружен непустой список;
- происходит запрос дополнительного объема памяти от распределителя страничного уровня. При этом создается нужное количество буферов заданного размера.

Алгоритм Мак-Кьюзика-Кэрелса (МКК)

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 перечисленных состояний:

- быть свободной. В этом случае соответствующий элемент массива содержит указатель на элемент, описывающий следующую свободную страницу;
- быть разбитой на буферы определенного размера. Элемент массива содержит размер буфера.

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 страниц. Отсутствие заголовка в выделенных буферах позволяет экономить память при удовлетворении запросов с потребностью в памяти, кратной некоторой степени числа 2.

Сравнение по фактору использования

Алгоритм блоков по 2 в степени n, при отсутствии блоков ближайшей степени 2, может выделить в 2 (а то и в 4) раза больше памяти, чем необходимо. В случае алгоритма МКК страница разбивается на нужные блоки во время выделения памяти (а не заранее), что позволяет распоряжаться ей более экономно. Следовательно, фактор использования алгоритмя блоков по 2 меньше, чем у алгоритма МКК.

Сравнение по простоте использования

В алгоритме блоков по 2 очень просто реализовывается выделение и освобождение, так как все, что нужно сделать:

- для выделения - взять блок равного или большего размера из нужного списка (или дать понять, что список таких блоков пуст);
- для освобождения - вернуть в список из заголовка.

А вот для МКК все немного сложнее. Для выделения нужно проверять, остались ли еще неиспользованные блоки. Если да, то выделить их (как в блоках по 2). Иначе, нужно разбить свободную страницу на нужного размера блоки и выделить один из них. Для освобождения нужно смотреть на размер блоков страницы и добавлять блок к списку соответствующего размера. Но все это позволяет избавится от заголовков для блоков и повысить фактор использования.

Код программы

alloc_MKK.hpp

```
#pragma once

#include <iostream>
#include <map>
#include <vector>
#include <cmath>

const size_t MAX_POW = 10;
const size_t MAX_BLOCK = 1024;

const size_t MIN_POW = 5;
const size_t MIN_BLOCK = 32;

const size_t PAGE_SIZE = 4096;

std::vector<size_t> pwrs = {1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096};

struct Page{
size_t _size;
void* _page_start;
};

class Allocator{
private:
std::vector<Page> _pages;
std::map<size_t, std::vector<void*>> _free_blocks;

public:
Allocator(size_t mem){
while(mem >= PAGE_SIZE){
```

```
Page tmp;
tmp._size = 0; // == free page
tmp._page_start = (void*)malloc(PAGE_SIZE);
if(tmp._page_start == NULL) {std::cerr << "malloc: error" << std::endl; exit(1);}
else _pages.push_back(tmp);
// std::cout << tmp._page_start << std::endl;
mem -= PAGE_SIZE;
}
// std::cout << _pages.size() << "\n";
}

void* alloc(size_t block_size){
if(block_size > PAGE_SIZE){std::cerr << "alloc: to_do" << std::endl; return NULL;}
void* res = NULL;
size_t power = ceil(log2(block_size));
block_size = pwrs[power];
// std::cout << block_size << std::endl;
bool get = false;
for(auto& elem : _pages){
if(elem._size == block_size){
if(!_free_blocks[block_size].empty()){
res = _free_blocks[block_size].back();
// std::cout << "Size of list " << block_size << " is " << _free_blocks[block_size].size() << std::endl;
_free_blocks[block_size].pop_back();
get = true;
break;
}
}
else if(elem._size == 0){
elem._size = block_size;
void* tmp = elem._page_start;
size_t count = PAGE_SIZE;
while(count > block_size){
_free_blocks[block_size].push_back(tmp);
tmp += block_size;
count -= block_size;
}
// std::cout << "Size of list " << block_size << " is " << _free_blocks[block_size].size() << std::endl;
res = tmp;
get = true;
break;
}
}
if(!get) {std::cerr << "alloc: no clear pages" << std::endl; exit(3);}
return res;
}
void free(void* block){
if(block == NULL) return;
bool get_free = false;
for(auto& elem : _pages){
if(block >= elem._page_start and block <= (elem._page_start + PAGE_SIZE)){
_free_blocks[elem._size].push_back(block);
```

```
get_free = true;
}
}
if(!get_free){
std::cerr << "No permission to free the 'block'" << std::endl;
}
}
};
```

alloc_pow2.hpp

```
#pragma once

#include <iostream>
#include <vector>
#include <map>
#include <cmath>


const size_t MAX_POW = 10;
const size_t MAX_BLOCK = 1024;

const size_t MIN_POW = 5;
const size_t MIN_BLOCK = 32;

std::vector<size_t> pwrs = {1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024};



class Allocator{
private:
std::map<size_t, std::vector<void*>> _free_blocks;
public:
Allocator(size_t memory_size){
std::vector<size_t> pwrs(MAX_POW + 1, 0);
size_t actual_max_power = MAX_POW;
size_t actual_size = 0;
size_t power = 0;
size_t actual_buf = 0;
while(memory_size > actual_size){

if(power){
actual_buf *= 2;
++power;
}else{
power = MIN_POW;
actual_buf = MIN_BLOCK;
}
if(memory_size < actual_size + actual_buf) actual_max_power = power;
else{
++pwrs[power];
actual_size += actual_buf + sizeof(size_t);
```

```
}

if(memory_size < actual_size + MIN_BLOCK) break;
power %= actual_max_power;
}

// // // // // // // // // //
// std::cout << "To check number of available 2_pow blocks (from 0 power to 10)\n";
// for(auto elem : pwrs){
// std::cout << elem << ' ';
// }
// std::cout << std::endl;
// // // // // // // // // //


size_t cur_block_size = MIN_BLOCK;
for(size_t i = MIN_POW; i <= MAX_POW; i++){
if(pwrs[i] == 0) break;

while(pwrs[i]--){
size_t* tmp;
tmp = (size_t*)malloc(cur_block_size);
if(tmp == NULL){
std::cerr << "malloc: error" << std::endl;
exit(1);
}
tmp[0] = i;
_free_blocks[i].push_back((void*)tmp);

// // // // // // // // // //
// std::cout << tmp << ' ' << (tmp + 1) << '\n';
// // // // // // // // // //
}
cur_block_size *= 2;
}
}

void* alloc(size_t block_size){
block_size += sizeof(size_t);
size_t power = ceil(log2(block_size));
power = std::max(power, MIN_POW);
while(power < MAX_POW and _free_blocks.find(power) == _free_blocks.end()){
power++;
}

void* res = NULL;
if(power <= MAX_POW and _free_blocks.find(power) != _free_blocks.end()){
if(!_free_blocks[power].empty()){
res = _free_blocks[power].back() + sizeof(size_t);
// std::cout << res << std::endl;
// std::cout << "Size of list " << power << " is " << allocator._free_blocks[power].size() << std::endl;
_free_blocks[power].pop_back();
```

```
}
}
// else{
// std::cerr << "No block with size of 'block_size'\n";
// }
return res;
}

void free(void* block){
if(block != NULL){
size_t* buf = (size_t*)block - 1;
size_t key = buf[0];
if(key >= MIN_POW or key <= MAX_POW or key%2 == 0){
_free_blocks[key].push_back(buf);
}
else{
std::cerr << "No permission to free the 'block'" << std::endl;
}
}
}

};
```

test.cpp

```
// #include "alloc_MKK.hpp"
#include "alloc_pow2.hpp"

#include <iostream>
#include <chrono>
#include <queue>
#include <vector>

const size_t MEM = 4000000;

using namespace std;

int main(){

vector<size_t> test_blocks;
size_t t;
while(cin >> t){
test_blocks.push_back(t);
}

// cout << "Test blocks count: " << test_blocks.size() << endl;

auto begin = chrono::high_resolution_clock::now();

Allocator All(MEM);

auto end = chrono::high_resolution_clock::now();
```

```
// cout << "Init test time (microseconds): " << chrono::duration_cast<chrono::microseconds>(end - begin).count()
<< endl;
queue<void*> Q;

begin = chrono::high_resolution_clock::now();

size_t null_count = 0;
for(auto elem : test_blocks){
void* tmp = All.alloc(elem);
// cout << tmp << endl;
if(tmp == NULL) null_count++;
else{
Q.push(tmp);
}
}

end = chrono::high_resolution_clock::now();

cout << "Allocation test time (microseconds): " << chrono::duration_cast<chrono::microseconds>(end -
begin).count() << endl;

begin = chrono::high_resolution_clock::now();

while(!Q.empty()){
All.free(Q.back());
Q.pop();
}

end = chrono::high_resolution_clock::now();

cout << "Free test time (microseconds): " << chrono::duration_cast<chrono::microseconds>(end - begin).count()
<< endl;

cout << "NULL count: " << null_count << endl;

return 0;
}
```

Протокол работы программы

В приведенных тестах значение размера памяти аллокаторов равняется 4000000 байт, файлы test и small_blocks_test представляют собой последовательности из 2001 элемента (размера блоков), запрашиваемых у аллокатора (рандомные и одного размера (32 байта)).

Тест выделения и освобождения для алгоритма блоков по 2 в степени n

```
mishazhadnov@McB-airmi build % ./test <../test.txt

Allocation test time (microseconds): 978
```

Free test time (microseconds): 253

NULL count: 35

mishazhadnov@McB-airmi build % ./test <../small_blocks_test.txt

Allocation test time (microseconds): 2252

Free test time (microseconds): 595

NULL count: 59

Тест выделения и освобождения для алгоритма Мак-Кьюзика-Кэрелса

mishazhadnov@McB-airmi build % ./test <../test.txt

Allocation test time (microseconds): 1390

Free test time (microseconds): 17065

NULL count: 0

mishazhadnov@McB-airmi build % ./test <../small_blocks_test.txt

Allocation test time (microseconds): 1412

Free test time (microseconds): 30521

NULL count: 0

Вывод

В ходе написания курсового проекта я подробно изучил алгоритмы аллокации блоков по 2 в степени n и Мак-Кьюзика-Кэрелса, на практике убедился в их достоинствах и недостатках. В будущем этот опыт поможет мне писать специализированные алгоритмы распределения памяти для более эффективного ее использования.