Національний технічний університет України

«Київський політехнічний інститут»

Факультет інформатики і обчислювальної техніки

Кафедра обчислювальної техніки

**Лабораторна робота №2**

**з курсу: «Сучасні операційні системи»**

*Виконав:*

студент групи ІC-72

Горохвський І.О.

*Перевірив:* Сімоненко А.В.

Київ, 2020 р.

**Тема**: Аллокатор пам’яті загального призначення (частина 1).

**Мета**: розробити аллокатор загального призначення.

# Краткая теория

В лабораторной работе №1 был рассмотрен простой метод построения аллокатора общего назначения. Рассмотрим другой метод решения этой же задачи. Далее описан один из вариантов реализации.

Вся виртуальная память состоит из страниц. Каждая страница может находиться в оперативной памяти или во внешнем файле (рассматриваются страницы, к которым было хотя бы одно обращение, т.е. страницы для которых были выделены ресурсы виртуальной памяти). При обращении к странице, которая находится во внешнем файле, происходит страничный промах и операционная система находит свободную физическую страницу и считывает в неё содержимое из внешнего файла. Размер страницы обычно равен от 4 Кбайт до нескольких Мбайт. Таким образом, если в процессе принятия решения аллокатор памяти обращается к меньшему числу страниц, тем меньше он оставляет след в памяти, тем он эффективнее.

Как и в лабораторной работе №1 аллокатор памяти запрашивает некоторую область памяти у операционной системы. Далее вся эта память делится на страницы. Размер страницы не обязательно должен совпадать с размером виртуальной страницы. Например, одна виртуальная может вмещать несколько страниц аллокатора. Все страницы выровнены, таким образом при обращении к данным по любому адресу внутри страницы мы обращаемся только к одной виртуальной странице.

Все блоки памяти делятся на две группы. В первую группу входят блоки с размерами меньше или равными половине страницы, во вторую все остальные. Блоки первой группы делятся на классы. Блоки одного класса имеют одинаковый размер. Например, этот размер может быть число 2x (x ≥ 4). Если приложение запрашивает блок памяти некоторого размера меньшего или равного половине странице, то аллокатор назначает этому блоку ближайший по размеру класс. Блоки второй группы это блоки размером в одну или несколько страниц. Если приложение запрашивает блок памяти некоторого размера большего чем половина страницы, то аллокатор округляет этот размер до ближайшего целого числа страниц.

Каждая страница может находиться в одном из трёх состояний: страница свободна, страница разделена на блоки одного класса или страница занята многостраничным блоком. Аллокатор содержит список свободных страниц. Этот список можно создать сразу после получения области памяти или по мере освобождения занятых страниц. Если страница разделена на блоки одного класса, то в этой странице могут быть только блоки этого класса и все блоки имеют одинаковый размер. Если страница занята многостраничным блоком, то после этой страницы могут находиться ноль или несколько страниц одного блока памяти.

Для каждой страницы существует описатель страницы, который однозначно определяет её состояние. Место нахождения описателя можно определить по указателю на описатель. Все указатели на описатели страниц можно разместить в одномерном массиве (длина массива равна числу страниц в выделенной области памяти). Для страницы разделённой на небольшие блоки описатель страницы можно хранить в самой странице. Для страницы разделённой на б*о*льшие блоки описатель страницы можно хранить в блоке меньшего размера (чтобы получить этот блок, необходимо рекурсивно вызвать аллокатор памяти). Для много страничных блоков описатели страниц не требуются, всю информацию о количестве страниц можно закодировать в указателях на описатели страниц многостраничного блока.

Для страницы разделённой на блоки в описателе страницы есть указатель на первый свободный блок в странице, а также счётчик количества свободных блоков в этой странице. Все свободные блоки в одной странице связаны в список, поля для связывания в список находятся в самих свободных блоках. При освобождении некоторого блока аллокатор по адресу блока высчитывает номер его страницы, по номеру страницы определяет указатель на описатель страницы, добавляет освобождённых блок в список свободных блоков и увеличивает счётчик свободных блоков. Если все блоки свободны, то эта страница освобождается.

Описатели страниц, разделённых на блоки одного и того же класса и у которых есть хотя бы один свободный блок, связаны в список. Страницы, в которых нет свободных блоков, в этот список не входят, но если у страницы появляется один свободный блок, то она добавляется в этот список. В аллокаторе памяти есть массив указателей на эти списки для каждого класса блоков. При запросе на выделение памяти проверяется список описателей страниц нужного класса. Если этот список пустой, то берётся пустая страница, разделяется на блоки и добавляется в список. Если список не пустой, то берётся первая в списке страница и в ней первый свободный блок отдаётся приложению.

Учёт свободной памяти производится при помощи дерева, где ключом является размер свободного пространства. Память для вершин дерева берётся в самих свободных страницах, составляющих свободные участки памяти.

В приведенном выше алгоритме массив указателей на описатели страниц не обязателен. Для страниц разделённых на небольшие блоки описатель страницы можно размещать прямо в странице (признаком того, что писатель страницы находится в начале странице, может быть адрес блока). Для всех остальных страниц описатели страниц можно размещать в хеш таблице, где ключом поиска является адрес страницы, память для хеш таблицы можно получить при помощи рекурсивного вызова аллокатора.

**Код програми:**

**PageVersion.cpp**

#include "PageVersion.h"  
  
using namespace std;  
  
Description Allocator::defineCategory(size\_t s){  
 Description pd;  
 if (s > (pageSize >> 1)){  
 pd.state = 2;  
 pd.bsize = s / pageSize;  
 pd.bsize += ((s % pageSize) > 0)? 1 : 0;  
 } else {  
 pd.state = 1;  
 pd.bsize = defineBlockSize(s);  
 }  
 return pd;  
}  
  
  
Allocator::Allocator(const size\_t n, const size\_t ps){  
 pages = n / ps;  
 size\_t pds = sizeof(Description)/sizeof(size\_t);  
 cout << pds << endl;  
  
 size\_t spaseForPD = pds\*pages;  
   
 size\_t maxbs = ps >> 1;  
 size\_t minbs = 16;  
 lBlocksLength = 1;  
 while(maxbs != minbs){  
 maxbs >>= 1;  
 lBlocksLength++;  
 }  
 size\_t needControlSpase = lBlocksLength + spaseForPD;  
  
 size\_t \*mas = new size\_t[n+needControlSpase];   
 size = n;  
 begin = mas;  
 pagesBegin = mas + needControlSpase;  
   
 pageSize = ps;  
 pageDescriptors = (Description\*)begin;  
 lBlocks = begin + spaseForPD;  
  
 for(int i = 0; i < pages; i++){  
 pageDescriptors[i].state = 0;  
 }  
 for(int i = 0; i < lBlocksLength; i++){  
 lBlocks[i] = pages+1;  
 }  
 initPages();  
 cout << begin << endl;  
 cout << pagesBegin << endl;  
 cout << size << endl;  
 cout << pageSize << endl;  
 cout << pages << endl;  
 cout << lBlocksLength << endl;  
 cout << pageDescriptors << endl;  
 cout << lBlocks << endl;  
}  
  
void\* Allocator::mem\_alloc(size\_t s){  
 Description def = defineCategory(s);  
 size\_t\* res;  
 if(def.state == 1){  
 res = getFreeLBlock(def.bsize);  
 } else {  
 res = getFreeMBlock(def.bsize);  
 }  
 return res;  
}  
  
  
size\_t Allocator::defineBlockSize(size\_t s){  
 size\_t res = 16;  
 while(s > res){  
 res <<=1;  
 }  
 return res;  
}  
  
size\_t\* Allocator::getFreeMBlock(size\_t ps){  
 Description pd = pageDescriptors[firstFreePage];  
 size\_t firstPage = firstFreePage;  
 size\_t thisPage = firstFreePage;  
 size\_t nextPage;  
 size\_t counter = 0;  
 while(pd.next <= pages && counter < ps){  
 nextPage = pd.next;  
 if(nextPage - 1 == thisPage){  
 counter++;   
 } else {  
 counter = 0;  
 firstPage = nextPage;  
 }  
 thisPage = nextPage;  
 pd = pageDescriptors[thisPage];  
 nextPage = pd.next;  
 }  
 if(counter == ps){  
 firstFreePage = nextPage;  
 return getAbsolutePageAddr(firstFreePage);  
 } else {  
 return NULL;  
 }  
}  
  
bool Allocator::freeLBlockIsLast(Description pd){  
 LeftBD\* desc = pd.firstFree;  
 if(desc->nextFreeBlock){  
 return false;  
 } else {  
 return true;  
 }  
}  
  
size\_t\* Allocator::getFreeLBlock(size\_t bs){  
 size\_t numberOfPage = lBlocks[getIndex(bs)];  
 size\_t index;  
 if(numberOfPage > pages){  
 index = createLBlockPage(bs);   
 } else {  
 index = numberOfPage;  
 }  
 Description pd = pageDescriptors[index];  
 LeftBD\* numberOfBlock = pd.firstFree;  
 if(numberOfBlock->nextFreeBlock == 0){  
 pd.firstFree = NULL;  
 lBlocks[index] = pages+1;  
 } else {  
 pd.firstFree = (LeftBD\*)((size\_t\*)(numberOfBlock) + numberOfBlock->nextFreeBlock);  
 }  
 return (size\_t\*)pd.firstFree;  
}  
  
size\_t Allocator::createLBlockPage(size\_t bs){  
 size\_t index = getFreePage();  
 if(index <= pages){  
 Description pd = pageDescriptors[index];  
 pd.state = 1;  
 pd.bsize = bs;  
 size\_t\* addr = getAbsolutePageAddr(index);  
 for(int i = 0; i < pages/bs; i++){  
 LeftBD\* des = (LeftBD\*)addr[i\*pd.bsize];  
 des->nextFreeBlock = 1;  
 if(i == pages/bs - 1)  
 des->nextFreeBlock = 0;  
 }  
 pd.firstFree = 0;  
 pd.next = pages+1;  
 lBlocks[getIndex(bs)] = index;  
 }  
 return index;  
}  
  
size\_t\* Allocator::getAbsolutePageAddr(size\_t index){  
 size\_t\* res = NULL;  
 if(index <= pages){  
 res = &(pagesBegin[pageSize\*index]);  
 }   
 return res;  
}  
  
size\_t Allocator::getFreePage(){  
 Description pd = pageDescriptors[firstFreePage];  
 firstFreePage = pd.next;  
 return pages+1;  
}  
  
void Allocator::setAllFree(Description pd){  
 size\_t bs = pd.bsize;  
 size\_t blocks = pageSize / bs;  
 for(int i = 0; i < blocks; i++){  
  
 }  
}  
  
void Allocator::initPages(){  
 for(int i = 0; i < pages; i++){  
 pageDescriptors[i].state = 0;  
 pageDescriptors[i].next = i+1;  
 }  
}  
  
size\_t Allocator::getIndex(size\_t s){  
 size\_t counter = 0;  
 while(s > 1){  
 counter++;  
 s >>= s;  
 }  
 counter -= 4;  
 return counter;  
}  
  
  
size\_t Allocator::findPageByAddress(size\_t\* addr){  
 size\_t shiftFromBegin = addr - pagesBegin;  
 size\_t pageNumber = shiftFromBegin / pageSize;  
 return pageNumber;  
}  
  
size\_t Allocator::findBlockByAddress(size\_t\* addr, size\_t bs){  
 size\_t shiftFromBegin = addr - pagesBegin;  
 size\_t shiftFromPageBegin = shiftFromBegin % pageSize;  
 size\_t number = shiftFromPageBegin / bs;  
 return number;  
}  
  
void Allocator::mem\_dump(){  
 Description pd;  
 for (int i = 0; i < pages; i++){  
 pd = pageDescriptors[i];  
 cout << "["<<i<<"] " << pd.state << " " << pd.bsize << " "  
 << pd.next << " " << pd.firstFree << endl;  
 }  
}  
  
void\* Allocator::mem\_realloc(void\* addr, size\_t size){  
 if(addr == NULL){  
 return mem\_alloc(size);  
 }  
 size\_t\* beg = (size\_t\*)addr;  
 size\_t pageNumber = findPageByAddress(beg);  
 PageDescriptor pd = pageDescriptors[pageNumber];  
 size\_t usefulMem;  
 if(pd.state == 1){  
 usefulMem = pd.bsize;  
 } else{  
 usefulMem = pd.bsize \* pageSize;  
 }  
  
 if(size == usefulMem){  
 return addr;  
 }  
 mem\_free(addr);  
 size\_t\* newAddr = (size\_t\*) mem\_alloc(size);  
 size\_t length = min(size, usefulMem);  
 copyData(beg, newAddr, length);  
 return newAddr;  
}  
  
void Allocator::copyData(size\_t\* from, size\_t\* to, size\_t length){  
 if (from > to){  
 for(int i = 0; i < length; i++){  
 to[i] = from[i];  
 }  
 } else {  
 for(int i = length-1; i <= 0; i++){  
 to[i] = from[i];  
 }  
 }  
}  
  
void Allocator::mem\_free(void\* addr){  
 size\_t pageNumber = findPageByAddress((size\_t\*)addr);  
 PageDescriptor pd = pageDescriptors[pageNumber];  
 if(pd.state == 1){  
 size\_t block = findBlockByAddress((size\_t\*)addr, pd.bsize);  
 if(pd.firstFree == NULL){  
 pd.firstFree = (LeftBD\*)(addr);  
 pd.firstFree->nextFreeBlock = 0;  
 } else {  
 size\_t shift = (size\_t\*)addr - (size\_t\*)(pd.firstFree);  
 pd.firstFree = (LeftBD\*)(addr);  
 pd.firstFree->nextFreeBlock = shift;  
 }  
 } else {  
 size\_t pgs = pd.bsize;  
 for(int i = 0; i < pgs; i++){  
 pd = pageDescriptors[pageNumber+i];  
 pd.state = 0;  
 pd.bsize = 0;  
 pd.firstFree = NULL;  
 pd.next = firstFreePage;  
 firstFreePage = pageNumber+i;  
 }  
 }  
}

**PageVersion.h**

#pragma once   
  
#include <Windows.h>  
#include <iostream>  
  
struct LeftBD{  
 size\_t nextFreeBlock;  
};  
  
struct Description{  
 size\_t next;  
 LeftBD\* firstFree;  
 size\_t bsize;  
 char state; //0-free, 1-left, 2-right  
};  
  
class Allocator {  
public:   
 Allocator(const size\_t ms, const size\_t ps);  
  
 void\* mem\_realloc(void \*addr, size\_t size);  
  
 void mem\_free(void \*addr);  
  
 void mem\_dump();  
  
 void\* mem\_alloc(size\_t size);  
  
   
private:  
 Description defineCategory(size\_t s);  
 size\_t defineBlockSize(size\_t s);  
 void initPages();  
 size\_t\* getFreeLBlock(size\_t bs);  
 size\_t getIndex(size\_t s);  
 size\_t createLBlockPage(size\_t bs);  
 size\_t getFreePage();  
 size\_t\* getAbsolutePageAddr(size\_t index);  
 void setAllFree(Description pd);  
 size\_t\* getFreeMBlock(size\_t ps);  
 bool freeLBlockIsLast(Description pd);  
 size\_t findPageByAddress(size\_t\* addr);  
 size\_t findBlockByAddress(size\_t\* addr, size\_t bs);  
 void copyData(size\_t\* from, size\_t\* to, size\_t length);  
 size\_t\* begin;  
 size\_t\* pagesBegin;  
 size\_t size;  
 size\_t pages;  
 size\_t pageSize;  
 Description\* pageDescriptors;  
 size\_t\* lBlocks;  
 size\_t lBlocksLength;  
 size\_t firstFreePage;  
};

**Test.cpp**

#include "PageVersion.h"  
  
using namespace std;  
  
int main(){  
 int n = 100;  
 size\_t size = 4194304;  
 size\_t ps = 4096;  
  
 Allocator al(size, ps);  
 size\_t\*\* addrArray = new size\_t\*[n];  
 for(int i = 0; i < n; i++){  
 addrArray[i] = (size\_t\*)al.mem\_alloc(rand());  
 if(addrArray[i] == NULL){  
 cout << "Error occurred" << endl;  
 }  
 }  
 al.mem\_dump();  
 for(int i = 0; i < n/3; i++){  
 al.mem\_realloc(addrArray[i], rand());  
 }  
  
 al.mem\_dump();  
 for(int i = n/2; i < n; i++){  
 al.mem\_free(addrArray[i]);  
 }  
 al.mem\_dump();  
 return 0;  
}