МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«Київський політехнічний інститут»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

Кафедра СКС

Лабораторна робота №2

з дисципліни

системне програмне забезпечення

на тему:

**«Аллокатор пам’яті загального призначення (частина 2)»**

Виконав студент Перевірив:

Групи КВ-64 \_\_\_\_\_\_\_\_\_\_\_

### Подольський Сергій Валентинович

залікова книжка № КВ6415 \_\_\_\_\_\_\_\_\_\_\_\_\_(бали)

#### VІІ семестр

Київ-2009

1. **Краткая теория и постановка задачи**

# 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.

## Задание на работу

См. лабораторную работу №1.

## Отчёт

См. лабораторную работу №1.

1. **Описание разработанного алгоритма.**

Алгоритм программы был целенаправлен на ускорение поиска свободного фрагмента при выделении памяти, а также на уменьшение фрагментации памяти с учётом разбиения её на страницы. Обе цели достигнуты следующим образом. При инициализации аллокатора создаётся три таблицы:

1. Таблица страниц, которые содержат в себе свободные блоки размером не больше половины страницы (многоблоковые страницы). Все свободные блоки в таблице распределены по классам. Блоки одного класса имеют одинаковый размер. Этот размер представляет собой число 2x, то есть размер, представляемый каждым следующим классом, в два раза больше. Все страницы одного класса связаны в список.
2. Таблица свободных блоков, которые состоят из одной или последовательности многих свободных страниц (многостраничные свободные блоки). Все эти блоки в таблице таким же образом распределены по классам и представляют собой, в отличие от первой таблицы, не страницы, связанные в список, а целые группы страниц, связанные в список. Следует отметить, что размер таких блоков, в отличие от первой таблицы, не всегда может быть одинаковым в пределах одного класса. Принадлежность свободного многостраничного блока памяти определённому классу устанавливается, если округление размера данного блока до ближайшей степени двойки соответствует размеру, за который отвечает этот класс. Кроме того, с целью поиска наиболее подходящего свободного блока памяти в каждом классе список блоков отсортирован в порядке возрастания их размеров. Таким образом, при попытке выделения блока памяти аллоктором сначала запрашиваемый размер округляется до ближайшей степени двойки, затем по полученному числу определяется класс свободных блоков, после чего список свободных блоков просматривается от начала до нахождения первого подходящего по размеру блока, то есть наиболее подходящего. Если же такой блок в данном классе свободных блоков не найден, аллокатор выдаёт первый блок следующего класса, содержащего блоки в два раза большего минимального размера. Если же все последующие классы не содержат свободных блоков вообще, аллокатор возвращает NULL.

Таким образом, вторая таблица является логическим продолжением первой, так как обе таблицы содержат классы свободных блоков памяти, однако первая таблица содержит классы многоблоковых страниц, а вторая – многостраничных свободных блоков. Строки первой таблицы содержат адреса дескрипторов страниц, а строки второй – адреса свободных страниц.

1. Таблица указателей на дескрипторы всех страниц. Каждому указателю взаимно однозначно ставится в соответствие одна страница. Количество указателей равно количеству страниц, указатели для каждой страницы следуют в порядке следования страниц. В каждом указателе закодирована информация о состоянии страницы. На дескрипторы ссылаются только те указатели, для которых определено состояние многоблоковой страницы.

Суммароное количество классов обеих таблиц рассчитывается так, чтобы один блок последнего класса (содержащего наибольшие свободные блоки) мог покрытьсвоим размером всю остальную неслужебную свободную область памяти.

Каждая страница может находиться в одном из четырёх состояний, которые кодируются двумя младшими битами указателя на дескриптор страницы:

|  |  |
| --- | --- |
| 11 | страница свободна |
| 10 | страница занята и не является последней в занятом многостраничном блоке памяти |
| 01 | страница занята и является последней в занятом многостраничном блоке памяти |
| 00 | страница разбита на 2N блоков одинакового размера |

Если два младших бита указателя на дескриптор равны нулю (страница является многоблоковой), то указатель содержит адрес дескриптора. Такой способ кодирования состояния страницы был введён с учётом того, что адрес дескриптора и так всегда кратен 4 и содержит нули в младших разрядах.

При освобождении многостраничного блока памяти освобожденный блок сливается со свободными соседними. Таким образом, из таблицы удаляются соседние свободные блоки выше и ниже текущего (если они есть), после чего результирующий свободный блок добавляется в соответствующий класс, не нарушая в нём упорядоченности блоков по размеру.

Оценим расход памяти для хранения служебной информации. В худшем случае

P \* (S div (P + L)) ≤ 2L\*(N-1) \* M

где S – общий размер доступной области памяти;

M – минимальный размер блока памяти (4 байта);

P – размер страницы (4к байт);

N – общее количество строк первой и второй таблиц;

L – размер строки каждой таблицы (4 байта).

Исходя из этого найдём количество строк таблиц, необходимое для хранения классов для заданной области памяти:

Количество указателей на дескрипторы страниц:

Общий размер служебной информации:

Из данного принципа работы алгоритма следует, что процент расхода памяти на служебную информацию не будет превышать

Время поиска свободного блока памяти сводится к поиску свободного блока внутри некоторого класса в таблице классов, так как адрес самого класса вычисляется по размеру запрашиваемой памяти. Для запрашиваемой области памяти, размер которой не больше половины страницы, берётся первая свободная страница памяти в классе, так как все страницы содержат блоки одинакового размера в рамках одного класса. А поскольку многостраничные блоки в классе упорядочены по возрастанию их размеров, то оценка поиска свободного многостраничного блока зависит от количества блоков меньшего размера, чем необходимо выделить, содержащихся в соответствующем классе.

Оценка времени освобождения блока памяти также различается при данном подходе для блоков размером не более половины страницы и для блоков, которые состоят из одной или больше страниц. В отличие от тривиального освобождения, заданный алгоритм требует добавления многостраничного блока в список с сохранением упорядоченности, что замедляет функцию при освобождении мультистраничных блоков.

Преимущества разработанного алгоритма:

* быстрый поиск свободного блока памяти;
* низкая фрагментация памяти.

Недостатки:

* незначительно замедлено время освобождения больших блоков памяти.

1. **Текст программы**

/\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* General-purpose Memory Allocator (part 2) \*

\* System software Lab №2 (1st semester 2009) \*

\* \*

\* Author: \*

\* Podolsky Sergey \*

\* Group: KV-64 \*

\* \*

\* written: 13.11.2009 \*

\* \*

\* Project definition: \*

\* Main.cpp The entry point for the console application \*

\* SlabAllocator.h Defines Memory Allocator class \*

\* stdafx.h Include file for standard system include files \*

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*/

#pragma once

#include "stdafx.h"

class SlabAllocator

{

/\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* Definitions \*

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*/

#define BLOCK\_TYPE void\*

#define SIZE\_TYPE unsigned int

#define SIZE\_LENGTH sizeof(SIZE\_TYPE)

#define MIN\_SIZE\_DEGREE 2

#define MIN\_BLOCK\_SIZE (1 << MIN\_SIZE\_DEGREE)

#define PAGE\_SIZE 4096

static const SIZE\_TYPE PAGE\_HALF = PAGE\_SIZE / 2;

/\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* Descriptor structure \*

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*/

#define DESCRIPTOR struct Descriptor

struct Descriptor

{

DESCRIPTOR\* next;

DESCRIPTOR\* prev;

SIZE\_TYPE total\_count;

SIZE\_TYPE free\_count;

SIZE\_TYPE\* first\_free\_block;

SIZE\_TYPE\* page;

};

/\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* Page states \*

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*/

#define FREE\_PAGE 3

#define LAST\_BLOCK\_PART 2

#define NOT\_LAST\_PART 1

#define MULTIBLOCK 0

#define STATE\_MASK 3

static const SIZE\_TYPE POINTER\_MASK = STATE\_MASK ^ 0xFFFFFFFF;

/\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* Local variables \*

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*/

SIZE\_TYPE\* block\_class\_table;

SIZE\_TYPE\* descriptor\_pointer\_table;

SIZE\_TYPE page\_count;

#define CLASS\_COUNT (descriptor\_pointer\_table - block\_class\_table)

#define PAGES\_OFFSET (descriptor\_pointer\_table + page\_count)

/\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* Local functions \*

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*/

static inline double log2(double x)

{

static const double l2 = 1.0 / log(2.0);

return log(x) \* l2;

}

static inline SIZE\_TYPE round\_size(SIZE\_TYPE size)

{

return size <= MIN\_BLOCK\_SIZE ? MIN\_BLOCK\_SIZE : 1 << (SIZE\_TYPE)ceil(log2(size));

}

static inline SIZE\_TYPE get\_page\_state(SIZE\_TYPE\* descriptor\_pointer)

{

return \*descriptor\_pointer & STATE\_MASK;

}

static void set\_page\_state(SIZE\_TYPE\* descriptor\_pointer, SIZE\_TYPE new\_state)

{

\*descriptor\_pointer &= POINTER\_MASK;

\*descriptor\_pointer |= new\_state;

}

inline SIZE\_TYPE\* get\_class\_by\_size(SIZE\_TYPE size)

{

return block\_class\_table + (size <= MIN\_BLOCK\_SIZE ? 0 : (SIZE\_TYPE)ceil(log2(size)) - MIN\_SIZE\_DEGREE);

}

inline DESCRIPTOR\* get\_page\_by\_descriptor\_pointer(SIZE\_TYPE\* descriptor\_pointer)

{

return (DESCRIPTOR\*)((SIZE\_TYPE)PAGES\_OFFSET + PAGE\_SIZE \* (descriptor\_pointer - descriptor\_pointer\_table));

}

inline SIZE\_TYPE\* get\_descriptor\_pointer\_by\_page(SIZE\_TYPE\* page)

{

return descriptor\_pointer\_table + ((SIZE\_TYPE)page - (SIZE\_TYPE)PAGES\_OFFSET) / PAGE\_SIZE;

}

DESCRIPTOR\* search\_free\_page\_sequence(SIZE\_TYPE required\_count)

{

for ( SIZE\_TYPE\* block\_class = get\_class\_by\_size(required\_count \* PAGE\_SIZE); block\_class < descriptor\_pointer\_table; block\_class++)

for ( DESCRIPTOR\* page\_sequence = ((DESCRIPTOR\*)block\_class)->next; page\_sequence != NULL; page\_sequence = page\_sequence->next )

if ( page\_sequence->total\_count >= required\_count )

return page\_sequence;

return NULL;

}

void add\_free\_page\_sequence\_to\_size\_class(DESCRIPTOR\* page\_sequence, SIZE\_TYPE page\_count)

{

// Set count of free pages

page\_sequence->total\_count = page\_count;

// Get size class

SIZE\_TYPE\* size\_class = get\_class\_by\_size(page\_count \* PAGE\_SIZE);

// Insert sequence into chain in order of ascending sequences

for ( DESCRIPTOR\* enumerator = (DESCRIPTOR\*)size\_class; ; enumerator = enumerator->next )

if ( enumerator->next == NULL || enumerator->next->total\_count >= page\_count )

{

page\_sequence->next = enumerator->next;

page\_sequence->prev = enumerator;

enumerator->next = page\_sequence;

if ( page\_sequence->next != NULL )

page\_sequence->next->prev = page\_sequence;

break;

}

}

void remove\_node\_from\_chain(DESCRIPTOR\* descriptor)

{

DESCRIPTOR \*next = descriptor->next, \*prev = descriptor->prev;

prev->next = next;

if ( next != NULL ) next->prev = prev;

}

SIZE\_TYPE\* allocate\_multipage\_block(SIZE\_TYPE size)

{

// Required page count

SIZE\_TYPE required\_count = (SIZE\_TYPE)ceil((double)size / PAGE\_SIZE);

// Get page sequence of required count

DESCRIPTOR\* page\_sequence = search\_free\_page\_sequence(required\_count);

if ( page\_sequence == NULL ) return NULL;

// Break chain

remove\_node\_from\_chain(page\_sequence);

// Set each page state to used

SIZE\_TYPE\* descriptor\_pointer = get\_descriptor\_pointer\_by\_page((SIZE\_TYPE\*)page\_sequence);

for ( SIZE\_TYPE i = 0; i < required\_count - 1; i++ )

set\_page\_state(descriptor\_pointer++, NOT\_LAST\_PART);

set\_page\_state(descriptor\_pointer++, LAST\_BLOCK\_PART);

// Count of free pages left

SIZE\_TYPE left\_count = ((DESCRIPTOR\*)page\_sequence)->total\_count - required\_count;

// If free pages left

if ( left\_count > 0 )

add\_free\_page\_sequence\_to\_size\_class(get\_page\_by\_descriptor\_pointer(descriptor\_pointer), left\_count);

// return address

return (SIZE\_TYPE\*)page\_sequence;

}

void free\_multipage\_block(SIZE\_TYPE\* descriptor\_pointer)

{

// Set each page state to free

SIZE\_TYPE\* enumerator = descriptor\_pointer;

while ( get\_page\_state(enumerator) == NOT\_LAST\_PART )

set\_page\_state(enumerator++, FREE\_PAGE);

set\_page\_state(enumerator++, FREE\_PAGE);

// Get free pages before current block

SIZE\_TYPE\* before = descriptor\_pointer;

while ( before - 1 >= descriptor\_pointer\_table && get\_page\_state(before - 1) == FREE\_PAGE )

before--;

// If there are free pages before then remove them from size class

if ( descriptor\_pointer - before > 0 )

remove\_node\_from\_chain(get\_page\_by\_descriptor\_pointer(before));

// Get free pages after current block

SIZE\_TYPE\* after = enumerator;

while ( after < PAGES\_OFFSET && get\_page\_state(after) == FREE\_PAGE )

after++;

// If there are free pages after then remove them from size class

if ( after - enumerator > 0 )

remove\_node\_from\_chain(get\_page\_by\_descriptor\_pointer(enumerator));

// Merge all free pages and add them to size class

add\_free\_page\_sequence\_to\_size\_class(get\_page\_by\_descriptor\_pointer(before), after - before);

}

SIZE\_TYPE\* allocate\_little\_block(SIZE\_TYPE size)

{

// Get block size class

SIZE\_TYPE\* size\_class = get\_class\_by\_size(size);

// If class contains free blocks

if ( \*size\_class != NULL )

{

// Get descriptor

DESCRIPTOR\* descriptor = (DESCRIPTOR\*)\*size\_class;

// Remove first block from free block list

SIZE\_TYPE\* first\_free\_block = descriptor->first\_free\_block;

descriptor->first\_free\_block = (SIZE\_TYPE\*)\*first\_free\_block;

// If page is full then remove it from size class

if ( --descriptor->free\_count == 0 )

remove\_node\_from\_chain(descriptor);

// return address

return first\_free\_block;

}

else

{

// Allocate single page

SIZE\_TYPE\* page = (SIZE\_TYPE\*)mem\_alloc(PAGE\_SIZE);

if ( page == NULL ) return NULL;

// Result

SIZE\_TYPE\* address;

// Descriptor

DESCRIPTOR\* descriptor;

// Round size as degree of 2

size = round\_size(size);

// If block size fits to descriptor size then create descriptor inside page

if ( round\_size(sizeof(DESCRIPTOR)) == size )

{

descriptor = (DESCRIPTOR\*)page;

// Set total block count

descriptor->total\_count = PAGE\_SIZE / size - 1;

// Set free block count

descriptor->free\_count = descriptor->total\_count - 1;

// Set first free block inside page

descriptor->first\_free\_block = (SIZE\_TYPE\*)((SIZE\_TYPE)page + 2 \* size);

// Set address

address = (SIZE\_TYPE\*)((SIZE\_TYPE)page + size);

}

else

{

// Allocate memory for descriptor

descriptor = (DESCRIPTOR\*)mem\_alloc(sizeof(DESCRIPTOR));

// If returned NULL then return NULL

if ( descriptor == NULL )

{

mem\_free(page);

return NULL;

}

// Set total block count

descriptor->total\_count = PAGE\_SIZE / size;

// Set first free block inside page

descriptor->first\_free\_block = (SIZE\_TYPE\*)((SIZE\_TYPE)page + size);

// Set free block count

descriptor->free\_count = descriptor->total\_count - 1;

// Set address

address = page;

}

// Set descriptor page

descriptor->page = page;

// Set descriptor pointer in descriptor table to descriptor address

\*get\_descriptor\_pointer\_by\_page(page) = (SIZE\_TYPE)(descriptor);

// Initialize free block list references

SIZE\_TYPE\* free\_block = descriptor->first\_free\_block;

for ( SIZE\_TYPE i = 0; i < descriptor->free\_count - 1; i++ )

free\_block = (SIZE\_TYPE\*)(\*free\_block = (SIZE\_TYPE)free\_block + size);

// Add page to block size class of one page that was found earlier

descriptor->next = (DESCRIPTOR\*)\*size\_class;

descriptor->prev = (DESCRIPTOR\*)size\_class;

\*size\_class = (SIZE\_TYPE)descriptor;

if ( descriptor->next != NULL )

descriptor->next->prev = descriptor;

// Return address

return address;

}

}

void free\_little\_block(SIZE\_TYPE\* block)

{

// Get descriptor pointer

SIZE\_TYPE\* descriptor\_pointer = get\_descriptor\_pointer\_by\_page((SIZE\_TYPE\*)block);

// Get page address

DESCRIPTOR\* page = get\_page\_by\_descriptor\_pointer(descriptor\_pointer);

// Get page descriptor

DESCRIPTOR\* descriptor = (DESCRIPTOR\*)\*descriptor\_pointer;

// If page is free

if ( ++descriptor->free\_count == descriptor->total\_count )

{

// Remove descriptor

if ( descriptor != page )

mem\_free(descriptor);

// Remove page from multiblock page list

remove\_node\_from\_chain(descriptor);

// Free page

set\_page\_state(descriptor\_pointer, LAST\_BLOCK\_PART);

mem\_free(page);

}

// If page was full before

else if ( descriptor->free\_count == 1 )

{

// Get block size class

SIZE\_TYPE\* size\_class = get\_class\_by\_size(PAGE\_SIZE / (descriptor == page ? descriptor->total\_count + 1 : descriptor->total\_count));

// Add page to block size class of one page that was found earlier

descriptor->next = (DESCRIPTOR\*)\*size\_class;

descriptor->prev = (DESCRIPTOR\*)size\_class;

\*size\_class = (SIZE\_TYPE)descriptor;

if ( descriptor->next != NULL )

descriptor->next->prev = descriptor;

}

}

/\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* Public methods \*

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*/

public:

/\*====================== Constructor ================================\*/

SlabAllocator(BLOCK\_TYPE memory\_space, SIZE\_TYPE memory\_size)

{

// Create table of block classes

descriptor\_pointer\_table = block\_class\_table = (SIZE\_TYPE\*)memory\_space;

for ( SIZE\_TYPE class\_size = MIN\_BLOCK\_SIZE / 2; class\_size < ((memory\_size - CLASS\_COUNT \* SIZE\_LENGTH) / (PAGE\_SIZE + SIZE\_LENGTH)) \* PAGE\_SIZE; class\_size \*= 2 )

\*(descriptor\_pointer\_table++) = NULL;

// Create table of descriptor pointers

page\_count = (memory\_size - CLASS\_COUNT \* SIZE\_LENGTH) / (PAGE\_SIZE + SIZE\_LENGTH);

for (SIZE\_TYPE i = 0; i < page\_count; i++)

set\_page\_state(descriptor\_pointer\_table + i, FREE\_PAGE);

// Add all pages to last (the biggest) class

block\_class\_table[CLASS\_COUNT - 1] = (SIZE\_TYPE)PAGES\_OFFSET;

DESCRIPTOR\* free\_sequence = (DESCRIPTOR\*)PAGES\_OFFSET;

free\_sequence->total\_count = page\_count;

free\_sequence->next = NULL;

free\_sequence->prev = (DESCRIPTOR\*)(block\_class\_table + CLASS\_COUNT - 1);

}

/\*====================== Allocate memory ============================\*/

BLOCK\_TYPE mem\_alloc(SIZE\_TYPE size)

{

if ( size > PAGE\_HALF )

return allocate\_multipage\_block(size);

else

return allocate\_little\_block(size);

}

/\*====================== Free memory ================================\*/

void mem\_free(BLOCK\_TYPE block)

{

// Get page descriptor pointer

SIZE\_TYPE\* descriptor\_pointer = get\_descriptor\_pointer\_by\_page((SIZE\_TYPE\*)block);

// If page state is block part then

if ( get\_page\_state(descriptor\_pointer) != MULTIBLOCK )

free\_multipage\_block(descriptor\_pointer);

else

free\_little\_block((SIZE\_TYPE\*)block);

}

/\*====================== Reallocate memory ==========================\*/

BLOCK\_TYPE mem\_realloc(BLOCK\_TYPE addr, SIZE\_TYPE new\_size)

{

if ( addr == NULL ) return mem\_alloc(new\_size);

SIZE\_TYPE\* old\_block = (SIZE\_TYPE\*)addr;

// Get page descriptor pointer

SIZE\_TYPE\* descriptor\_pointer = get\_descriptor\_pointer\_by\_page(old\_block);

// Get old size

SIZE\_TYPE old\_size = 1;

if ( get\_page\_state(descriptor\_pointer) == MULTIBLOCK )

{

DESCRIPTOR\* descriptor = (DESCRIPTOR\*)\*descriptor\_pointer;

old\_size = PAGE\_SIZE / ((SIZE\_TYPE\*)descriptor == descriptor->page ? descriptor->total\_count + 1 : descriptor->total\_count);

}

else

{

while ( get\_page\_state(descriptor\_pointer++) == NOT\_LAST\_PART )

old\_size++;

old\_size \*= PAGE\_SIZE;

}

// If size is the same then return current block

if ( old\_size == (new\_size <= PAGE\_HALF ? round\_size(new\_size) : PAGE\_SIZE \* (SIZE\_TYPE)ceil((double)new\_size / PAGE\_SIZE)) ) return old\_block;

// Allocate block of new size

SIZE\_TYPE\* new\_block = (SIZE\_TYPE\*)mem\_alloc(new\_size);

// If could not allocate then return NULL

if ( new\_block == NULL ) return NULL;

// Copy info from old block to new one

memcpy(new\_block, old\_block, min(old\_size, new\_size));

// Free old block

mem\_free(old\_block);

// Return new block

return new\_block;

}

/\*====================== Dump each page state =======================\*/

void mem\_dump()

{

for ( SIZE\_TYPE i = 0; i < page\_count; i++ )

{

SIZE\_TYPE state = get\_page\_state(descriptor\_pointer\_table + i);

cout << get\_page\_by\_descriptor\_pointer(descriptor\_pointer\_table + i) << ":\t" <<

(state == FREE\_PAGE ? "Free page" : state == MULTIBLOCK ? "Used multiblock page" : state == LAST\_BLOCK\_PART ? "Used block part (last page)" : "Used block part (not last page)") << endl;

}

cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << endl;

}

/\*====================== Dump table with all free blocks ============\*/

void table\_dump()

{

SIZE\_TYPE\* block\_class = block\_class\_table;

for ( SIZE\_TYPE size = MIN\_BLOCK\_SIZE; block\_class < descriptor\_pointer\_table; size \*= 2, block\_class++ )

{

cout << size << " bytes:" << endl;

if ( size > PAGE\_HALF )

for ( DESCRIPTOR\* page\_sequence = ((DESCRIPTOR\*)block\_class)->next; page\_sequence != NULL; page\_sequence = page\_sequence->next )

cout << "\t" << page\_sequence << ":\t" << page\_sequence->total\_count << " free pages in free block\n";

else

for ( DESCRIPTOR\* descriptor = (DESCRIPTOR\*)\*block\_class; descriptor != NULL; descriptor = descriptor->next )

cout << "\t" << descriptor->page << ":\t" << descriptor->free\_count << "/" << descriptor->total\_count << " free blocks in page\n";

}

cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << endl;

}

};

1. **Пример работы аллокатора**

**1) Код:**

SIZE\_TYPE size = 20000;

SlabAllocator\* allocator = new SlabAllocator(malloc(size), size);

allocator->mem\_dump();

allocator->table\_dump();

**Результат:**

004F2004: Free page

004F3004: Free page

004F4004: Free page

004F5004: Free page

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

4 bytes:

8 bytes:

16 bytes:

32 bytes:

64 bytes:

128 bytes:

256 bytes:

512 bytes:

1024 bytes:

2048 bytes:

4096 bytes:

8192 bytes:

16384 bytes:

004F2004: 4 free pages in free block

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Press any key to continue . . .

**Пояснение:**

Размер страницы – 4к, минимальный размер блока – 4 байта. Аллокатору даётся область памяти, которой хватит на 4 страницы и служебную информацию в виде таблиц.

Выведены адрес и состояние каждой страницы из таблицы указателей на дескрипторы страниц (все страницы свободны), а также содержание каждого класса свободных блоков из обеих таблиц классов. Вся память представляет собой единственный свободный блок, который содержится в последнем классе. Все остальные классы пустые.

**2) Код:**

SIZE\_TYPE size = 5 \* 4096 + 1000; // 5 pages and 1000 bytes for tables

SlabAllocator\* allocator = new SlabAllocator(malloc(size), size);

void\* block1 = allocator->mem\_alloc(5000);

for (int i = 0; i < 1024 + 1; i++)

allocator->mem\_alloc(4);

allocator->mem\_dump();

allocator->table\_dump();

**Результат:**

0029200C: Used block part (not last page)

0029300C: Used block part (last page)

0029400C: Used multiblock page

0029500C: Used multiblock page

0029600C: Used multiblock page

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

4 bytes:

0029600C: 1023/1024 free blocks in page

8 bytes:

16 bytes:

32 bytes:

0029500C: 125/127 free blocks in page

64 bytes:

128 bytes:

256 bytes:

512 bytes:

1024 bytes:

2048 bytes:

4096 bytes:

8192 bytes:

16384 bytes:

32768 bytes:

**Пояснение:**

Для выделения памяти при первом вызове понадобилось две страницы, поэтому первые две страницы заняты многостраничным блоком. Далее циклически запрашивается выделение маленьких блоков памяти по 4 байта. Для выделения заданного количества маленьких блоков понадобилось разбить на маленькие блоки две страницы. Абсолютно все блоки первой страницы заняты, поэтому эта страница не содержится в классе свободных блоков. Однако во второй странице занят только 1 блок из 128 доступных (4 \* 128 = 4к), поэтому она содержится в классе сободных блоков размером до 4 байт. Для разбиения двох страниц на блоки понадобилось создать для них два дескриптора, каждый из которых занимает до 32 байт. Для этих двух дескрипторов выделена отдельно страница и разбита на блоки по 32 байта. Дескриптор для данной страницы хранится в самой странице, поэтому общее количество блоков в ней меньше на 1 и равно 127, а не 128. Поскольку страница содержит 2 дескриптора для других страниц, то 125 блоков в ней являются свободными и страница содержится в классе свободных блоков размером до 32 байт.