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: М8О-215Б-23

Студент: Верменников М.В.

Преподаватель: Миронов Е.С.

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата: 26.12.24

Москва, 2024

**Постановка задачи**

**Вариант 16.**

Необходимо сравнить два алгоритма аллокации: алгоритм Мак-Кьюзи-Кэрелся и блоки по 2 в степени n 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.

**Общий метод и алгоритм решения**

Использованные системные вызовы:

* mmap(void \*addr, size\_t length, int prot, int flags, int fd, off\_t offset) – выделяет непрерывный регион памяти (обычно анонимная маппинг, без файла), возвращает указатель на него.
* munmap(void \*addr, size\_t length) – освобождает регион памяти, ранее отображённый через mmap.

Цель данной лабораторной работы сравнить две разные стратегии выделения памяти на примере написания двух аллокаторов: аллокатора, использующего список свободных блоков, и аллокатора, использующего метод двойников. Рассмотрим подробнее каждый из алгоритмом, а потом сравним их на различных тестах.

**Аллокатор McKuzi основан на поддержке одного или нескольких списков свободных блоков, а также механизме «best fit». При запуске программы ему передаётся крупная непрерывная область памяти, которую он инициализирует единственным большим свободным блоком. Этот блок описывается управляющим заголовком, в котором хранится информация о размере блока, признаке занятости (свободен ли он в данный момент) и указателе на следующий блок в связном списке.**

**Когда аллокатору McKuzi поступает запрос на выделение памяти, он сканирует свой список свободных блоков и подбирает блок, который наилучшим образом соответствует запрошенному размеру. При этом выбирается блок, у которого «запас» (разница между размером блока и запрошенным размером) оказывается минимально достаточным, то есть это и есть принцип «best fit». Если выбранный блок заметно больше, чем необходимо, то для уменьшения фрагментации аллокатор разделяет его на два: один блок точно по запрошенному размеру, а оставшаяся часть оформляется как новый свободный блок и добавляется в общий список. Указатель, возвращаемый пользователю, сдвинут за границу управляющего заголовка, чтобы пользователь работал только с полезными байтами.**

**Освобождение памяти в McKuzi сводится к тому, что по переданному указателю вычисляется адрес заголовка и он помечается как свободный. Далее аллокатор старается объединить его с соседними свободными блоками (смотрит, не лежит ли рядом с ним другой свободный блок). Это объединение, называемое coalescing, позволяет избежать накопления множества мелких фрагментов и даёт более крупные свободные области. Сильной стороной такого подхода оказывается относительно высокий «фактор использования» памяти, поскольку при правильной реализации «best fit» и механизма слияния фрагментация уменьшается. С другой стороны, поиск подходящего блока в линейном списке на каждом вызове может становиться довольно затратным, если число блоков велико.**

**Аллокатор Pow2 работает по принципу, предполагающему использование блоков памяти, размеры которых являются степенями двойки. В классическом виде это бывает buddy-аллокатор, однако в упрощённой версии, на которой основан данный пример, область памяти может быть попросту разбита (или вообще не разбита) на блоки-кратные 2^N. При инициализации аллокатор получает большой регион (например, 1 ГБ), находит в нём подходящую степень двойки, которую может целиком отдать под управление, и выставляет соответствующие структуры для учёта свободных блоков.**

**Когда пользователь запрашивает некоторый размер S, аллокатор Pow2 округляет его (с учётом управляющих данных) до ближайшей степени двойки и пытается взять уже готовый блок из массива или списка свободных блоков этого размера. Если такой свободный блок есть, выдача происходит мгновенно. Если же нет, то в упрощённом варианте аллокатор может сообщить об исчерпании памяти или обратиться к списку более крупного размера (что тоже не всегда возможно). При освобождении памяти Pow2 возвращает освободившийся блок в его список, не всегда выполняя слияние. Из-за отсутствия полноценного механизма «дробления» и «объединения» (как в классическом buddy) может возникать значительная фрагментация.**

**Таким образом, Pow2 быстро выполняет операции `alloc` и `free`, зато легко «запирает» большую часть памяти в одном крупном блоке, тогда как McKuzi, несмотря на медленную работу при массовом выделении, обычно распоряжается своими свободными участками эффективнее.**

**Код программы**

**main.cpp**

#include<sys/mman.h>

#include<chrono>

#include<vector>

#include<thread>

#include<random>

#include<iostream>

// Подключаем наши заголовки с двумя реализациями

#include"Mckuzi.h"

#include"Pow2Allocator.h"

intmain()

{

std::cout <<"Creating allocators..\n";

// Общий объём памяти, отведённый под оба аллокатора

size\_tsize =1024\*1024\*1024;// 1 Гб, как в примере

// MAP\_PRIVATE | MAP\_ANONYMOUS: анонимная память, изменения видны только текущему процессу

void\*mem1 =mmap(nullptr, size, PROT\_READ |PROT\_WRITE, MAP\_PRIVATE |MAP\_ANONYMOUS, -1, 0);

void\*mem2 =mmap(nullptr, size, PROT\_READ |PROT\_WRITE, MAP\_PRIVATE |MAP\_ANONYMOUS, -1, 0);

// Создаём объекты аллокаторов

Allocator\_McKuzi al1(mem1, size);

Allocator\_Pow2 al2(mem2, size);

// Первичная проверка свободной памяти

std::cout <<"All memory: "

<<al1.get\_free\_memory() <<" (McKuzi), "

<<al2.get\_free\_memory() <<" (Pow2)\n";

std::cout <<"Created\n\n";

// -------------------

// TEST 1: Много маленьких блоков одного размера

// -------------------

{

std::cout <<"TEST 1 -- allocate lots of memory of a single little size\n";

intcycl\_count =5;

std::cout <<"Cycles count = "<<cycl\_count <<"\n\n";

intblocks\_count =10000;

// В примере: [1, 4, 8, 16, 32, 64],

// но мы можем сделать аналогично с «1 << t1\_sz[i]»

// (т.е. на самом деле это размеры 2^1, 2^4, 2^8 ...).

// Для наглядности оставим так же, как в примере:

intt1\_sz[5] ={1,4,8,16,32};

for(inti =0; i <cycl\_count; i++)

{

std::cout <<"Block size = 2^"<<t1\_sz[i]

<<" (="<<(1<<t1\_sz[i]) <<")"

<<", Block count = "<<blocks\_count <<std::endl;

// -- Выделяем blocks\_count блоков аллокатором al1

std::vector<void\*>ptrs1(blocks\_count);

autostart =std::chrono::high\_resolution\_clock::now();

for(intj =0; j <blocks\_count; j++)

{

ptrs1[j] =al1.alloc(1<<t1\_sz[i]);

}

autoend =std::chrono::high\_resolution\_clock::now();

autoduration1 =std::chrono::duration\_cast<std::chrono::microseconds>(end -start);

// -- Аналогично al2

std::vector<void\*>ptrs2(blocks\_count);

start =std::chrono::high\_resolution\_clock::now();

for(intj =0; j <blocks\_count; j++)

{

ptrs2[j] =al2.alloc(1<<t1\_sz[i]);

}

end =std::chrono::high\_resolution\_clock::now();

autoduration2 =std::chrono::duration\_cast<std::chrono::microseconds>(end -start);

// -- Освобождаем в аллокаторе al1

start =std::chrono::high\_resolution\_clock::now();

for(intj =0; j <blocks\_count; j++)

{

al1.free(ptrs1[j]);

}

end =std::chrono::high\_resolution\_clock::now();

autoduration3 =std::chrono::duration\_cast<std::chrono::microseconds>(end -start);

// -- И в аллокаторе al2

start =std::chrono::high\_resolution\_clock::now();

for(intj =0; j <blocks\_count; j++)

{

al2.free(ptrs2[j]);

}

end =std::chrono::high\_resolution\_clock::now();

autoduration4 =std::chrono::duration\_cast<std::chrono::microseconds>(end -start);

std::cout <<"Result Allocate:\n"

<<" McKuzi: "<<duration1.count() <<" us, "

<<" Pow2: "<<duration2.count() <<" us, ratio="

<<(duration2.count() ?double(duration1.count())/duration2.count() :0.0)

<<"\n";

std::cout <<"Result Free:\n"

<<" McKuzi: "<<duration3.count() <<" us, "

<<" Pow2: "<<duration4.count() <<" us, ratio="

<<(duration4.count() ?double(duration3.count())/duration4.count() :0.0)

<<"\n\n";

}

}

// -------------------

// TEST 2: Выделение памяти при сильной фрагментации

// -------------------

{

std::cout <<"TEST 2 -- allocate a lot of memory in the face of high fragmentation\n";

intblocks\_count =10000;

std::cout <<"Allocate "<<blocks\_count

<<" blocks of size=100 (McKuzi & Pow2)\n";

// -- аллокация для al1

std::vector<void\*>frament\_blocks1(blocks\_count);

for(inti =0; i <blocks\_count; ++i)

{

frament\_blocks1[i] =al1.alloc(100);

}

std::cout <<"Free all even blocks (al1)\n";

for(inti =0; i <blocks\_count; ++i)

{

if(i %2==0)// Освобождаем чётные

al1.free(frament\_blocks1[i]);

}

// -- аналогично для al2

std::vector<void\*>frament\_blocks2(blocks\_count);

for(inti =0; i <blocks\_count; ++i)

{

frament\_blocks2[i] =al2.alloc(100);

}

std::cout <<"Free all even blocks (al2)\n";

for(inti =0; i <blocks\_count; ++i)

{

if(i %2==0)

al2.free(frament\_blocks2[i]);

}

std::cout <<"Fragmentation completed\n";

// Теперь пробуем выделить (blocks\_count/2) блоков по 80

inthalf =blocks\_count /2;

std::cout <<"Block size=80, Block count="<<half <<std::endl;

// -- alloc

std::vector<void\*>ptrs1(half);

autostart =std::chrono::high\_resolution\_clock::now();

for(intj =0; j <half; j++)

{

ptrs1[j] =al1.alloc(80);

}

autoend =std::chrono::high\_resolution\_clock::now();

autoduration1 =std::chrono::duration\_cast<std::chrono::microseconds>(end -start);

std::vector<void\*>ptrs2(half);

start =std::chrono::high\_resolution\_clock::now();

for(intj =0; j <half; j++)

{

ptrs2[j] =al2.alloc(80);

}

end =std::chrono::high\_resolution\_clock::now();

autoduration2 =std::chrono::duration\_cast<std::chrono::microseconds>(end -start);

// -- free

start =std::chrono::high\_resolution\_clock::now();

for(intj =0; j <half; j++)

{

al1.free(ptrs1[j]);

}

end =std::chrono::high\_resolution\_clock::now();

autoduration3 =std::chrono::duration\_cast<std::chrono::microseconds>(end -start);

start =std::chrono::high\_resolution\_clock::now();

for(intj =0; j <half; j++)

{

al2.free(ptrs2[j]);

}

end =std::chrono::high\_resolution\_clock::now();

autoduration4 =std::chrono::duration\_cast<std::chrono::microseconds>(end -start);

std::cout <<"Result Allocate:\n"

<<" McKuzi: "<<duration1.count() <<" us, "

<<" Pow2: "<<duration2.count() <<" us, ratio="

<<(duration2.count() ?double(duration1.count())/duration2.count() :0.0)

<<"\n";

std::cout <<"Result Free:\n"

<<" McKuzi: "<<duration3.count() <<" us, "

<<" Pow2: "<<duration4.count() <<" us, ratio="

<<(duration4.count() ?double(duration3.count())/duration4.count() :0.0)

<<"\n\n";

// Наконец, полностью освобождаем неосвобождённые блоки

for(inti =0; i <blocks\_count; ++i)

{

if(i %2!=0)// Освобождаем только нечетные (чётные мы уже освободили)

{

al1.free(frament\_blocks1[i]);

al2.free(frament\_blocks2[i]);

}

}

}

// -------------------

// TEST 3: Случайные размеры блоков

// -------------------

{

std::cout <<"TEST 3 -- allocate random blocks\n";

intnum\_iterations =5;

intnum\_blocks =10000;// Use a lot of blocks

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<>distrib(4, 1024);// 4..1024

for(intiter =0; iter <num\_iterations; ++iter)

{

std::vector<size\_t>sizes(num\_blocks);

std::vector<void\*>ptrs1(num\_blocks);

std::vector<void\*>ptrs2(num\_blocks);

// генерируем размеры

for(inti =0; i <num\_blocks; ++i)

{

sizes[i] =distrib(gen);

}

// -- alloc McKuzi

autostart =std::chrono::high\_resolution\_clock::now();

for(inti =0; i <num\_blocks; ++i)

{

ptrs1[i] =al1.alloc(sizes[i]);

}

autoend =std::chrono::high\_resolution\_clock::now();

autoduration1 =std::chrono::duration\_cast<std::chrono::microseconds>(end -start);

// -- alloc Pow2

start =std::chrono::high\_resolution\_clock::now();

for(inti =0; i <num\_blocks; ++i)

{

ptrs2[i] =al2.alloc(sizes[i]);

}

end =std::chrono::high\_resolution\_clock::now();

autoduration2 =std::chrono::duration\_cast<std::chrono::microseconds>(end -start);

// -- free McKuzi

start =std::chrono::high\_resolution\_clock::now();

for(inti =0; i <num\_blocks; ++i)

{

al1.free(ptrs1[i]);

}

end =std::chrono::high\_resolution\_clock::now();

autoduration3 =std::chrono::duration\_cast<std::chrono::microseconds>(end -start);

// -- free Pow2

start =std::chrono::high\_resolution\_clock::now();

for(inti =0; i <num\_blocks; ++i)

{

al2.free(ptrs2[i]);

}

end =std::chrono::high\_resolution\_clock::now();

autoduration4 =std::chrono::duration\_cast<std::chrono::microseconds>(end -start);

std::cout <<"Iteration "<<(iter+1) <<":\n"

<<" Allocate => McKuzi: "<<duration1.count() <<" us, "

<<"Pow2: "<<duration2.count() <<" us, ratio="

<<(duration2.count() ?double(duration1.count())/duration2.count() :0.0) <<"\n"

<<" Free => McKuzi: "<<duration3.count() <<" us, "

<<"Pow2: "<<duration4.count() <<" us, ratio="

<<(duration4.count() ?double(duration3.count())/duration4.count() :0.0) <<"\n\n";

}

}

// -------------------

// TEST 4: Usage factor (McKuzi)

// -------------------

{

std::cout <<"TEST 4 -- Usage factor (McKuzi)\n";

constintnum\_requests =10000;

constsize\_tmax\_block\_size =512;

std::uniform\_int\_distribution<size\_t>dist(16, max\_block\_size);

std::random\_device rd;

std::mt19937 gen(rd());

size\_ttotal\_allocated =0;

size\_tmemory\_size =al1.get\_free\_memory();

std::cout <<"All memory (McKuzi) = "<<memory\_size <<'\n';

std::vector<void\*>ptrs(num\_requests);

for(inti =0; i <num\_requests; ++i)

{

size\_treq\_size =dist(gen);

void\*p =al1.alloc(req\_size);

ptrs[i] =p;

if(p) {

total\_allocated +=req\_size;

}

}

doubleutilization\_factor =0.0;

size\_tfree\_mem\_now =al1.get\_free\_memory();

if((memory\_size -free\_mem\_now) !=0)

{

utilization\_factor =double(total\_allocated) /double(memory\_size -free\_mem\_now);

}

std::cout <<"Total allocated: "<<total\_allocated <<" bytes\n"

<<"Free memory: "<<free\_mem\_now <<" bytes\n"

<<"Utilization factor: "<<utilization\_factor <<std::endl;

// Освобождаем

for(inti =0; i <num\_requests; ++i)

{

al1.free(ptrs[i]);

}

}

// -------------------

// TEST 5: Usage factor (Pow2)

// -------------------

{

std::cout <<"TEST 5 -- Usage factor (Pow2)\n";

constintnum\_requests =10000;

constsize\_tmax\_block\_size =512;

std::uniform\_int\_distribution<size\_t>dist(16, max\_block\_size);

std::random\_device rd;

std::mt19937 gen(rd());

size\_ttotal\_allocated =0;

size\_tmemory\_size =al2.get\_free\_memory();

std::cout <<"All memory (Pow2) = "<<memory\_size <<'\n';

std::vector<void\*>ptrs(num\_requests);

for(inti =0; i <num\_requests; ++i)

{

size\_treq\_size =dist(gen);

void\*p =al2.alloc(req\_size);

ptrs[i] =p;

if(p) {

total\_allocated +=req\_size;

}

}

doubleutilization\_factor =0.0;

size\_tfree\_mem\_now =al2.get\_free\_memory();

if((memory\_size -free\_mem\_now) !=0)

{

utilization\_factor =double(total\_allocated) /double(memory\_size -free\_mem\_now);

}

std::cout <<"Total allocated: "<<total\_allocated <<" bytes\n"

<<"Free memory: "<<free\_mem\_now <<" bytes\n"

<<"Utilization factor: "<<utilization\_factor <<std::endl;

// Освобождаем

for(inti =0; i <num\_requests; ++i)

{

al2.free(ptrs[i]);

}

}

// Освобождаем выделенную под аллокаторы память

munmap(mem1, size);

munmap(mem2, size);

return0;

}

**Mckuzi.h**

#pragmaonce

#include<cstddef> // size\_t

#include<iostream>

// Простейший вариант «Мак–Кьюзи–Кэрелс» можно сделать как “best fit” на списке

// или как несколько списков (segregated lists) — здесь для наглядности взят

// один список + поиск лучшего блока (best fit). Это не строгое каноническое

// воплощение Мак–Кьюзи–Кэрелса, но иллюстрирует подход.

// Структура описывает управляющий блок и память

classAllocator\_McKuzi

{

structBlock

{

size\_tsize; // размер данного блока (без учёта заголовка)

boolfree; // свободен ли блок

Block\*next; // односвязный список

};

private:

Block\*head;// Голова списка

public:

// Конструктор: realMemory — указатель на область памяти (mmap),

// memory\_size — её полный размер

Allocator\_McKuzi(void\*realMemory, size\_tmemory\_size);

// Выделение памяти

void\*alloc(size\_tblock\_size);

// Освобождение памяти

voidfree(void\*ptr);

// Подсчитать сколько всего свободно байт

size\_tget\_free\_memory();

};

**Mckuzi.cpp**

#include"Mckuzi.h"

Allocator\_McKuzi::Allocator\_McKuzi(void\*realMemory, size\_tmemory\_size)

{

// Инициализируем один большой блок

head=reinterpret\_cast<Block\*>(realMemory);

head->size=memory\_size-sizeof(Block);

head->free=true;

head->next=nullptr;

}

void\*Allocator\_McKuzi::alloc(size\_tblock\_size)

{

// Ищем блок с минимальным «избытком» (best fit)

Block\*bestBlock=nullptr;

Block\*prevBest =nullptr;// предыдущий для bestBlock

Block\*cur =head;

Block\*prev =nullptr;

size\_tminDiff=(size\_t)-1;// очень большое число

// 1) Найдём блок best fit

while(cur!=nullptr)

{

if(cur->free&&cur->size>=block\_size)

{

size\_tdiff=cur->size-block\_size;

if(diff<minDiff)

{

minDiff=diff;

bestBlock=cur;

prevBest =prev;

}

}

prev=cur;

cur =cur->next;

}

// Если подходящего блока не нашлось

if(!bestBlock)

{

returnnullptr;

}

// 2) Попробуем разделить (если достаточно места для заголовка + данных)

if(bestBlock->size>=block\_size+sizeof(Block) +8)

{

// Создаём новый блок «хвост» из избытка

char\*basePtr=reinterpret\_cast<char\*>(bestBlock);

Block\*newBlock=reinterpret\_cast<Block\*>( basePtr+sizeof(Block) +block\_size);

newBlock->size=bestBlock->size-block\_size-sizeof(Block);

newBlock->free=true;

newBlock->next=bestBlock->next;

// Изменяем размер у блока, который отдаём

bestBlock->size=block\_size;

bestBlock->free=false;

bestBlock->next=newBlock;

}

else

{

// Не делим — просто делаем блок занятым

bestBlock->free=false;

}

// 3) Возвращаем указатель на «полезную» область (после заголовка)

returnreinterpret\_cast<void\*>(bestBlock+1);

}

voidAllocator\_McKuzi::free(void\*ptr)

{

if(!ptr) return;

// Получаем заголовок

Block\*block=reinterpret\_cast<Block\*>(ptr) -1;

block->free=true;

// Попробуем объединять блоки (coalescing) с соседями

// Для этого пройдёмся с начала списка

Block\*cur=head;

Block\*prev=nullptr;

while(cur)

{

if(cur==block)

{

// Попробуем слить с предыдущим

if(prev&&prev->free)

{

// Объединяем prev и cur

prev->size+=sizeof(Block) +cur->size;

prev->next=cur->next;

cur=prev;// текущий блок расширен

}

// Попробуем слить с следующим

if(cur->next&&cur->next->free)

{

Block\*tmp=cur->next;

cur->size+=sizeof(Block) +tmp->size;

cur->next=tmp->next;

}

break;// закончили объединение

}

prev=cur;

cur =cur->next;

}

}

size\_tAllocator\_McKuzi::get\_free\_memory()

{

size\_ttotalFree=0;

Block\*cur=head;

while(cur)

{

if(cur->free)

{

totalFree+=cur->size;

}

cur=cur->next;

}

returntotalFree;

}

**Pow2Allocator.h**

#pragmaonce

#include<cstddef>

#include<iostream>

#include<vector>

// Упрощённый вариант «блоки по 2^n»:

// - Допустим, что минимальный блок = 2^MIN\_POW, максимальный = 2^MAX\_POW

// - Храним массив списков свободных блоков: freeLists[i] для блоков размера 2^(i)

// - При alloc находим 2^k >= (block\_size + заголовок), выдаём блок из freeLists[k]

// (если там пусто — попробуйте больший k).

// - При free возвращаем блок обратно в соответствующий список (без слияния).

// - Реальное объединение/деление для 2^n обычно приводит к buddy-аллокатору.

// Здесь, для простоты, «без объединения», иначе это будет почти Buddy :)

classAllocator\_Pow2

{

structHeader{

size\_tsizePow; // степень двойки, которая описывает размер этого блока

bool used; // метка занятости (для отладки, не обязательно)

};

private:

staticconstintMIN\_POW=4; // минимальный блок 16 байт (можно настроить)

staticconstintMAX\_POW=30;// верхняя граница ~1 Гб, чтобы не вылезти

// Массив списков (в виде односвязных цепочек)

// freeLists[k] хранит указатели на блоки размером 2^k

void\*freeLists[MAX\_POW+1];

// Указатель на начало всей области

char\*bufferStart;

size\_ttotalSize;

public:

Allocator\_Pow2(void\*realMemory, size\_tmemory\_size);

void\*alloc(size\_tblock\_size);

void free(void\*ptr);

size\_tget\_free\_memory();

private:

// Вспомогательные методы

int findPow2(size\_tneedSize);

void initMemory();

size\_tblockSizeFromPow(intp) { return(size\_t(1) <<p); }

};

**Pow2Allocator.cpp**

#include"Pow2Allocator.h"

#include<cmath>

#include<cstring>

// Конструктор

Allocator\_Pow2::Allocator\_Pow2(void\*realMemory, size\_tmemory\_size)

{

bufferStart=reinterpret\_cast<char\*>(realMemory);

totalSize =memory\_size;

// Изначально обнуляем все списки

for(inti=0; i<=MAX\_POW; i++) {

freeLists[i] =nullptr;

}

// Раз уж мы «разбиваем» всё пространство на блоки 2^n,

// упрощённо — проинициализируем его целиком списками по максимально большому блоку,

// затем можем «нарезать» вручную. Но для примера можно всё отдать как один большой блок

// ровно 2^P, где P — ближайшая вниз степень, чтобы не усложнять логику разбиения.

// 1) Находим самую большую степень двойки, которая <= memory\_size:

intbigP=MAX\_POW;

while( bigP>=MIN\_POW&&( (size\_t(1)<<bigP) >memory\_size) ) {

bigP--;

}

// Если уж совсем ничего не влезает, останавливаемся

if(bigP<MIN\_POW) bigP=MIN\_POW;

// Создаём один большой блок 2^bigP

Header\*h=reinterpret\_cast<Header\*>(bufferStart);

h->sizePow=bigP;

h->used =false;

// Следующий «указатель» в односвязном списке хранить будем в самом блоке

// просто через reinterpret\_cast<Header\*\*>(...) и т.п.

// Но для простоты пока сделаем так, что «адрес следующего» лежит после заголовка.

// freeLists[bigP] = (void\*)((char\*)h + sizeof(Header)); // вариант 1

freeLists[bigP] =h;// вариант 2: будем трактовать сам Header\* как голову списка

}

intAllocator\_Pow2::findPow2(size\_tneedSize)

{

// Ищем степень p, чтобы 2^p >= needSize

// Можно сделать быстро через built-in \_\_builtin\_clz и т.д., здесь сделаем цикл

intp=MIN\_POW;

while(p<=MAX\_POW)

{

if( (size\_t(1)<<p) >=needSize)

returnp;

p++;

}

return-1;// не нашли подходящий размер

}

void\*Allocator\_Pow2::alloc(size\_tblock\_size)

{

// + заголовок

size\_ttotalNeeded=block\_size+sizeof(Header);

intp=findPow2(totalNeeded);

if(p<0) {

returnnullptr;// слишком большой запрос

}

// Ищем любую свободную «цепочку» начиная с p и выше

inttryP=p;

for(; tryP<=MAX\_POW; tryP++)

{

if(freeLists[tryP] !=nullptr)

{

// Берём голову списка

Header\*blockHead=reinterpret\_cast<Header\*>( freeLists[tryP] );

// Убираем из freeLists[tryP]

freeLists[tryP] =nullptr;// т.к. мы хранить «один блок» для примера

// (в реальном коде, если список из многих, надо брать первый элемент

// и перенастраивать указатель списка на следующий)

blockHead->used=true;// пометим, что занято

// Если блок слишком большой и нужно расколоть — в упрощённом варианте

// мы можем не делать этого (или сделать на уровне buddy).

// Для демонстрации не будем.

// Просто вернём этот большой блок. Реально: тут бы раскалывать вниз

// до нужного p, но это уже buddy-аллокатор.

return(void\*)(blockHead+1);

}

}

// не нашли свободных блоков подходящего размера

returnnullptr;

}

voidAllocator\_Pow2::free(void\*ptr)

{

if(!ptr) return;

Header\*h=reinterpret\_cast<Header\*>(ptr) -1;

h->used=false;

// Возвращаем в список freeLists[h->sizePow],

// но у нас сейчас хранится «один блок» на список. Для наглядности:

// Если там что-то уже есть, мы его потеряем… В реальном коде надо делать из этого

// Linked list. Здесь — сверх-упрощённый вариант.

if(freeLists[h->sizePow] ==nullptr)

{

freeLists[h->sizePow] =h;

}

else

{

freeLists[h->sizePow] =h;

}

}

size\_tAllocator\_Pow2::get\_free\_memory()

{

// Пройдёмся по всем freeLists[] и суммируем

size\_ttotalFree=0;

for(inti=MIN\_POW; i<=MAX\_POW; i++)

{

if(freeLists[i] !=nullptr)

{

// Если хотим полноценный список — надо бежать по цепочке

// Здесь считаем, что в каждом freeLists[i] может лежать только один блок

Header\*h=reinterpret\_cast<Header\*>(freeLists[i]);

if(!h->used)

{

// размер блока = 2^i

totalFree+=(size\_t(1) <<i);

}

}

}

returntotalFree;

}

**Протокол работы программы**

**Тестирование:**

**Creating allocators...**

All memory: 1073741800 (McKuzi), 1073741824 (Pow2)

Created

TEST 1 -- allocate lots of memory of a single little size

Cycles count = 5

Block size = 2^1 (=2), Block count = 10000

Result Allocate:

McKuzi: 89898 us, Pow2: 173 us, ratio=519.642

Result Free:

McKuzi: 122 us, Pow2: 24 us, ratio=5.08333

Block size = 2^4 (=16), Block count = 10000

Result Allocate:

McKuzi: 96426 us, Pow2: 156 us, ratio=618.115

Result Free:

McKuzi: 142 us, Pow2: 24 us, ratio=5.91667

Block size = 2^8 (=256), Block count = 10000

Result Allocate:

McKuzi: 179731 us, Pow2: 187 us, ratio=961.128

Result Free:

McKuzi: 125 us, Pow2: 24 us, ratio=5.20833

Block size = 2^16 (=65536), Block count = 10000

Result Allocate:

McKuzi: 3942249 us, Pow2: 277 us, ratio=14231.9

Result Free:

McKuzi: 1223 us, Pow2: 47 us, ratio=26.0213

Block size = 2^32 (=1), Block count = 10000

Result Allocate:

McKuzi: 111821 us, Pow2: 209 us, ratio=535.029

Result Free:

McKuzi: 73 us, Pow2: 24 us, ratio=3.04167

TEST 2 -- allocate a lot of memory in the face of high fragmentation

Allocate 10000 blocks of size=100 (McKuzi & Pow2)

Free all even blocks (al1)

Free all even blocks (al2)

Fragmentation completed

Block size=80, Block count=5000

Result Allocate:

McKuzi: 237309 us, Pow2: 78 us, ratio=3042.42

Result Free:

McKuzi: 108800 us, Pow2: 14 us, ratio=7771.43

TEST 3 -- allocate random blocks

Iteration 1:

Allocate => McKuzi: 1105691 us, Pow2: 218 us, ratio=5071.98

Free => McKuzi: 178 us, Pow2: 28 us, ratio=6.35714

Iteration 2:

Allocate => McKuzi: 860009 us, Pow2: 220 us, ratio=3909.13

Free => McKuzi: 131 us, Pow2: 24 us, ratio=5.45833

Iteration 3:

Allocate => McKuzi: 849478 us, Pow2: 221 us, ratio=3843.79

Free => McKuzi: 164 us, Pow2: 24 us, ratio=6.83333

Iteration 4:

Allocate => McKuzi: 783380 us, Pow2: 174 us, ratio=4502.18

Free => McKuzi: 119 us, Pow2: 18 us, ratio=6.61111

Iteration 5:

Allocate => McKuzi: 593030 us, Pow2: 164 us, ratio=3616.04

Free => McKuzi: 110 us, Pow2: 17 us, ratio=6.47059

TEST 4 -- Usage factor (McKuzi)

All memory (McKuzi) = 1073741800

Total allocated: 2669122 bytes

Free memory: 1070832678 bytes

Utilization factor: 0.917501

TEST 5 -- Usage factor (Pow2)

All memory (Pow2) = 1073741824

Total allocated: 38 bytes

Free memory: 0 bytes

Utilization factor: 3.53903e-08

**Strace:**

**4276 execve("./allocator\_compare", ["./allocator\_compare"], 0x7fff708be798 /\* 26 vars \*/) = 0**

**4276 brk(NULL) = 0x1ad3d000**

**4276 mmap(NULL, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7f91a9a9d000**

**4276 access("/etc/ld.so.preload", R\_OK) = -1 ENOENT (No such file or directory)**

**4276 openat(AT\_FDCWD, "/etc/ld.so.cache", O\_RDONLY|O\_CLOEXEC) = 3**

**4276 newfstatat(3, "", {st\_mode=S\_IFREG|0644, st\_size=25786, ...}, AT\_EMPTY\_PATH) = 0**

**4276 mmap(NULL, 25786, PROT\_READ, MAP\_PRIVATE, 3, 0) = 0x7f91a9a96000**

**4276 close(3) = 0**

**4276 openat(AT\_FDCWD, "/usr/local/lib64/libstdc++.so.6", O\_RDONLY|O\_CLOEXEC) = 3**

**4276 read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0\0"..., 832) = 832**

**4276 newfstatat(3, "", {st\_mode=S\_IFREG|0755, st\_size=2530008, ...}, AT\_EMPTY\_PATH) = 0**

**4276 mmap(NULL, 2543808, PROT\_READ, MAP\_PRIVATE|MAP\_DENYWRITE, 3, 0) = 0x7f91a9828000**

**4276 mmap(0x7f91a98cd000, 1216512, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0xa5000) = 0x7f91a98cd000**

**4276 mmap(0x7f91a99f6000, 581632, PROT\_READ, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x1ce000) = 0x7f91a99f6000**

**4276 mmap(0x7f91a9a84000, 57344, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x25c000) = 0x7f91a9a84000**

**4276 mmap(0x7f91a9a92000, 12480, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_ANONYMOUS, -1, 0) = 0x7f91a9a92000**

**4276 close(3) = 0**

**4276 openat(AT\_FDCWD, "/lib/x86\_64-linux-gnu/libm.so.6", O\_RDONLY|O\_CLOEXEC) = 3**

**4276 read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0\0"..., 832) = 832**

**4276 newfstatat(3, "", {st\_mode=S\_IFREG|0644, st\_size=907784, ...}, AT\_EMPTY\_PATH) = 0**

**4276 mmap(NULL, 909560, PROT\_READ, MAP\_PRIVATE|MAP\_DENYWRITE, 3, 0) = 0x7f91a9749000**

**4276 mmap(0x7f91a9759000, 471040, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x10000) = 0x7f91a9759000**

**276 mmap(0x7f91a97cc000, 368640, PROT\_READ, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x83000) = 0x7f91a97cc000**

**4276 mmap(0x7f91a9826000, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0xdc000) = 0x7f91a9826000**

**4276 close(3) = 0**

**4276 openat(AT\_FDCWD, "/usr/local/lib64/libgcc\_s.so.1", O\_RDONLY|O\_CLOEXEC) = 3**

**4276 read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0\0"..., 832) = 832**

**4276 newfstatat(3, "", {st\_mode=S\_IFREG|0644, st\_size=906528, ...}, AT\_EMPTY\_PATH) = 0**

**4276 mmap(NULL, 181160, PROT\_READ, MAP\_PRIVATE|MAP\_DENYWRITE, 3, 0) = 0x7f91a971c000**

**4276 mmap(0x7f91a9720000, 143360, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x4000) = 0x7f91a9720000**

**4276 mmap(0x7f91a9743000, 16384, PROT\_READ, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x27000) = 0x7f91a9743000**

**4276 mmap(0x7f91a9747000, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x2b000) = 0x7f91a9747000**

**4276 close(3) = 0**

**4276 openat(AT\_FDCWD, "/lib/x86\_64-linux-gnu/libc.so.6", O\_RDONLY|O\_CLOEXEC) = 3**

**4276 read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20t\2\0\0\0\0\0"..., 832) = 832**

**4276 pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784**

**4276 newfstatat(3, "", {st\_mode=S\_IFREG|0755, st\_size=1922136, ...}, AT\_EMPTY\_PATH) = 0**

**4276 pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784**

**4276 mmap(NULL, 1970000, PROT\_READ, MAP\_PRIVATE|MAP\_DENYWRITE, 3, 0) = 0x7f91a953b000**

**4276 mmap(0x7f91a9561000, 1396736, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x26000) = 0x7f91a9561000**

**4276 mmap(0x7f91a96b6000, 339968, PROT\_READ, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x17b000) = 0x7f91a96b6000**

**4276 mmap(0x7f91a9709000, 24576, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x1ce000) = 0x7f91a9709000**

**4276 mmap(0x7f91a970f000, 53072, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_ANONYMOUS, -1, 0) = 0x7f91a970f000**

**4276 close(3) = 0**

**4276 mmap(NULL, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7f91a9539000**

**4276 arch\_prctl(ARCH\_SET\_FS, 0x7f91a953a480) = 0**

**4276 set\_tid\_address(0x7f91a953a750) = 4276**

**4276 set\_robust\_list(0x7f91a953a760, 24) = 0**

**4276 rseq(0x7f91a953ada0, 0x20, 0, 0x53053053) = 0**

**4276 mprotect(0x7f91a9709000, 16384, PROT\_READ) = 0**

**4276 mprotect(0x7f91a9747000, 4096, PROT\_READ) = 0**

**4276 mprotect(0x7f91a9826000, 4096, PROT\_READ) = 0**

**4276 mmap(NULL, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7f91a9537000**

**4276 mprotect(0x7f91a9a84000, 45056, PROT\_READ) = 0**

**4276 mprotect(0x405000, 4096, PROT\_READ) = 0**

**4276 mprotect(0x7f91a9acf000, 8192, PROT\_READ) = 0**

**4276 prlimit64(0, RLIMIT\_STACK, NULL, {rlim\_cur=8192\*1024, rlim\_max=RLIM64\_INFINITY}) = 0**

**4276 munmap(0x7f91a9a96000, 25786) = 0**

**4276 futex(0x7f91a9a9273c, FUTEX\_WAKE\_PRIVATE, 2147483647) = 0**

**4276 getrandom("\xd6\xf4\x18\xdb\xc3\xb7\x0b\x53", 8, GRND\_NONBLOCK) = 8**

**4276 brk(NULL) = 0x1ad3d000**

**4276 brk(0x1ad5e000) = 0x1ad5e000**

**4276 newfstatat(1, "", {st\_mode=S\_IFCHR|0620, st\_rdev=makedev(0x88, 0x1), ...}, AT\_EMPTY\_PATH) = 0**

**4276 write(1, "Creating allocators..\n", 22) = 22**

**4276 mmap(NULL, 1073741824, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7f9169537000**

**4276 mmap(NULL, 1073741824, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7f9129537000**

**4276 write(1, "All memory: 1073741800 (McKuzi),"..., 51) = 51**

**4276 write(1, "Created\n\n", 9) = 9**

**4276 write(1, "TEST 1 -- allocate lots of memor"..., 58) = 58**

**4276 write(1, "Cycles count = 5\n\n", 18) = 18**

**4276 write(1, "Block size = 2^1 (=2), Block cou"..., 43) = 43**

**4276 brk(0x1ad83000) = 0x1ad83000**

**4276 write(1, "Result Allocate:\n", 17) = 17**

**4276 write(1, " McKuzi: 92078 us, Pow2: 157 "..., 51) = 51**

**4276 write(1, "Result Free:\n", 13) = 13**

**4276 write(1, " McKuzi: 121 us, Pow2: 24 us,"..., 49) = 49**

**4276 brk(0x1ad70000) = 0x1ad70000**

**4276 write(1, "Block size = 2^4 (=16), Block co"..., 44) = 44**

**4276 brk(0x1ad97000) = 0x1ad97000**

**4276 write(1, "Result Allocate:\n", 17) = 17**

**4276 write(1, " McKuzi: 92253 us, Pow2: 185 "..., 51) = 51**

**4276 write(1, "Result Free:\n", 13) = 13**

**4276 write(1, " McKuzi: 65 us, Pow2: 24 us, "..., 48) = 48**

**4276 brk(0x1ad83000) = 0x1ad83000**

**4276 brk(0x1ad70000) = 0x1ad70000**

**4276 write(1, "Block size = 2^8 (=256), Block c"..., 45) = 45**

**4276 brk(0x1ad97000) = 0x1ad97000**

**4276 write(1, "Result Allocate:\n", 17) = 17**

**4276 write(1, " McKuzi: 184898 us, Pow2: 242"..., 52) = 52**

**4276 write(1, "Result Free:\n", 13) = 13**

**4276 write(1, " McKuzi: 453 us, Pow2: 32 us,"..., 49) = 49**

**4276 brk(0x1ad83000) = 0x1ad83000**

**4276 brk(0x1ad70000) = 0x1ad70000**

**4276 write(1, "Block size = 2^16 (=65536), Bloc"..., 48) = 48**

**4276 brk(0x1ad97000) = 0x1ad97000**

**4276 write(1, "Result Allocate:\n", 17) = 17**

**4276 write(1, " McKuzi: 3275186 us, Pow2: 27"..., 51) = 51**

**4276 write(1, "Result Free:\n", 13) = 13**

**4276 write(1, " McKuzi: 599 us, Pow2: 25 us,"..., 47) = 47**

**4276 brk(0x1ad83000) = 0x1ad83000**

**4276 brk(0x1ad70000) = 0x1ad70000**

**4276 write(1, "Block size = 2^32 (=1), Block co"..., 44) = 44**

**4276 brk(0x1ad97000) = 0x1ad97000**

**4276 write(1, "Result Allocate:\n", 17) = 17**

**4276 write(1, " McKuzi: 94777 us, Pow2: 191 "..., 51) = 51**

**4276 write(1, "Result Free:\n", 13) = 13**

**4276 write(1, " McKuzi: 55 us, Pow2: 24 us, "..., 48) = 48**

**4276 brk(0x1ad83000) = 0x1ad83000**

**4276 brk(0x1ad70000) = 0x1ad70000**

**4276 write(1, "TEST 2 -- allocate a lot of memo"..., 69) = 69**

**4276 write(1, "Allocate 10000 blocks of size=10"..., 50) = 50**

**4276 write(1, "Free all even blocks (al1)\n", 27) = 27**

**4276 brk(0x1ad97000) = 0x1ad97000**

**4276 write(1, "Free all even blocks (al2)\n", 27) = 27**

**4276 write(1, "Fragmentation completed\n", 24) = 24**

**4276 write(1, "Block size=80, Block count=5000\n", 32) = 32**

**4276 write(1, "Result Allocate:\n", 17) = 17**

**4276 write(1, " McKuzi: 226869 us, Pow2: 106"..., 51) = 51**

**4276 write(1, "Result Free:\n", 13) = 13**

**4276 write(1, " McKuzi: 109390 us, Pow2: 13 "..., 51) = 51**

**4276 brk(0x1ad83000) = 0x1ad83000**

**4276 brk(0x1ad70000) = 0x1ad70000**

**4276 write(1, "TEST 3 -- allocate random blocks"..., 33) = 33**

**4276 brk(0x1ad97000) = 0x1ad97000**

**4276 write(1, "Iteration 1:\n", 13) = 13**

**4276 write(1, " Allocate => McKuzi: 851621 us, "..., 60) = 60**

**4276 write(1, " Free => McKuzi: 133 us, Pow"..., 57) = 57**

**4276 brk(0x1ad83000) = 0x1ad83000**

**4276 brk(0x1ad70000) = 0x1ad70000**

**4276 brk(0x1ad97000) = 0x1ad97000**

**4276 write(1, "Iteration 2:\n", 13) = 13**

**4276 write(1, " Allocate => McKuzi: 880286 us, "..., 60) = 60**

**4276 write(1, " Free => McKuzi: 158 us, Pow"..., 58) = 58**

**4276 brk(0x1ad83000) = 0x1ad83000**

**4276 brk(0x1ad70000) = 0x1ad70000**

**4276 brk(0x1ad97000) = 0x1ad97000**

**4276 write(1, "Iteration 3:\n", 13) = 13**

**4276 write(1, " Allocate => McKuzi: 840714 us, "..., 60) = 60**

**4276 write(1, " Free => McKuzi: 134 us, Pow"..., 57) = 57**

**4276 brk(0x1ad83000) = 0x1ad83000**

**4276 brk(0x1ad70000) = 0x1ad70000**

**4276 brk(0x1ad97000) = 0x1ad97000**

**4276 write(1, "Iteration 4:\n", 13) = 13**

**4276 write(1, " Allocate => McKuzi: 840328 us, "..., 60) = 60**

**4276 write(1, " Free => McKuzi: 156 us, Pow"..., 53) = 53**

**4276 brk(0x1ad83000) = 0x1ad83000**

**4276 brk(0x1ad70000) = 0x1ad70000**

**4276 brk(0x1ad97000) = 0x1ad97000**

**4276 write(1, "Iteration 5:\n", 13) = 13**

**4276 write(1, " Allocate => McKuzi: 747553 us, "..., 60) = 60**

**4276 write(1, " Free => McKuzi: 145 us, Pow"..., 57) = 57**

**4276 brk(0x1ad83000) = 0x1ad83000**

**4276 brk(0x1ad70000) = 0x1ad70000**

**4276 write(1, "TEST 4 -- Usage factor (McKuzi)\n", 32) = 32**

**4276 write(1, "All memory (McKuzi) = 1073741800"..., 33) = 33**

**4276 write(1, "Total allocated: 2637125 bytes\n", 31) = 31**

**4276 write(1, "Free memory: 1070864675 byte"..., 34) = 34**

**4276 write(1, "Utilization factor: 0.916583\n", 29) = 29**

**4276 write(1, "TEST 5 -- Usage factor (Pow2)\n", 30) = 30**

**4276 write(1, "All memory (Pow2) = 1073741824\n", 31) = 31**

**4276 write(1, "Total allocated: 300 bytes\n", 27) = 27**

**4276 write(1, "Free memory: 0 bytes\n", 25) = 25**

**4276 write(1, "Utilization factor: 2.79397e-07\n", 32) = 32**

**4276 munmap(0x7f9169537000, 1073741824) = 0**

**4276 munmap(0x7f9129537000, 1073741824) = 0**

**4276 exit\_group(0) = ?**

**4276 +++ exited with 0 +++**

**Результат сравнения**

Для оценки эффективности этих двух аллокаторов (McKuzi и Pow2) были проведены несколько тестов, моделирующих различные сценарии использования памяти. Среди них:

* Выделение множества одинаковых небольших блоков подряд.
* Выделение блоков в условиях высокой фрагментации (половина ранее выделенных блоков освобождена).
* Выделение и освобождение блоков случайных размеров.
* Измерение «фактора использования памяти» (utilization factor) в ситуациях, когда память активно запрашивается случайными размерами.

Такой набор тестов позволяет увидеть сильные и слабые стороны каждого алгоритма. Результаты приведены ниже.

* **Фактор использования**

Фактор использования показывает, насколько эффективно реальный объём памяти задействован «под данные» по сравнению с общей зарезервированной памятью. Ниже таблица с итогами «Usage factor» для каждого из аллокаторов после серии запросов (**тест 4/5**).

|  |  |  |
| --- | --- | --- |
| Allocator | Mckuzi | Pow2 |
| All memory | 1073741800 | 1073741824 |
| Total allocated | 2669122 | 38 |
| Free memory | 1070832678 | 0 |
| Utilization factor | 0.917501 | 3.53903e-08 |

**McKuzi:** судя по результатам, суммарно пользователь запросил около 2.6 МБ, при этом в системе учтено ~3 МБ занятого (с учётом внутренних структур), остальная часть (примерно 1070 МБ) остаётся свободной. Коэффициент ~0.92 означает, что почти все выделенные байты действительно «работают на программу».

**Pow2:** в упрощённой реализации «блоков 2^n» всего один большой блок занимается при первом же запросе. Фактически все 1 ГБ признаётся занятой памятью (хотя нужно всего 38 байт). Отсюда гигантский объём «пустующего» пространства и экстремально малое значение фактора использования (~3.5 × 10^-8).

* **Скорость выделения и освобождения блоков**

Другие ключевые показатели – это время alloc и free. Ниже приведены выдержки из результатов нескольких тестов.

В **Тесте 1** последовательно выделялись и освобождались блоки заданного размера (2, 16, 256, 65536 и даже «2^32=1» из-за артефактов в примере). Для каждого размера брали 10 000 блоков. Пример фрагмента результатов:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Block size | Allocate (Mckuzi) / us | Allocate (Pow2) / us | Free (Mckuzi) / us | Free (Pow2) us |
| 2 | 89898 | 173 | 122 | 24 |
| 16 | 96426 | 156 | 142 | 24 |
| 256 | 179731 | 187 | 125 | 24 |
| 65536 | 3942249 | 277 | 1223 | 47 |

Как видно, **Pow2** в разы (а иногда и на порядок) быстрее при alloc, тогда как у **McKuzi** время может доходить до миллионов микросекунд при работе с крупными блоками. Это объясняется «линейным» поиском подходящего места в списке (и иногда «best fit»), приводящим к большим накладным расходам.

В **Тесте 2** сначала выделяли 10 000 блоков по 100 байт, освобождали половину из них (все чётные), и после этого пытались выделить ещё 5 000 блоков по 80 байт, чтобы посмотреть влияние «дырок» на производительность:

|  |  |  |
| --- | --- | --- |
| Allocator | Mckuzi  us | Pow2  us |
| Time alloc | 237309 | 78 |
| Time free | 108800 | 14 |

Алгоритм «Pow2» фактически не «видит» фрагментацию — он уже занял большой кусок, поэтому дополнительные небольшие alloc/free операций не меняют его структуру (в данной упрощённой версии). Результаты показывают колоссальный разрыв: McKuzi тратит на порядок больше времени.

В **Тесте 3** (5 итераций) берётся 10 000 блоков, их размеры распределены равномерно в диапазоне 4…1024 байт. Снова наблюдается аналогичная картина: Pow2 почти мгновенно обслуживает запросы (порядка нескольких сотен микросекунд на 10 000 блоков), в то время как McKuzi уходит за сотни тысяч или миллионы микросекунд:

Пример

Iteration 1:

* Allocate => McKuzi ~1 105 691 µs, Pow2 218 µs
* Free => McKuzi 178 µs, Pow2 28 µs

И так далее. Во всех итерациях аллокатор McKuzi работает заметно дольше при выделении.

* **Простота использования аллокатора**
* **McKuzi** (список + best fit)
  + Логика относительно проста: список свободных блоков, при alloc ищем наилучший блок, при free пытаемся «склеить» соседние свободные.
  + Главный плюс – неплохой «фактор использования» (особенно если блоки по размеру близки к реальным запросам). Минус – высокая временная сложность при больших массивах выделений (линейный поиск).
* **Pow2** (упрощённая «2^n»)
  + Фактически мгновенное время alloc/free в тестах, потому что вся память сразу разделена (или вообще не разделяется) на крупный блок, и мы не делаем сложных операций поиска.
  + Но такая простота ведёт к ужасной фрагментации и практически нулевому коэффициенту использования.

Если взять «настоящий buddy-allocator» (как в другом примере), он может и выделять быстро, и частично утилизировать память лучше. Но в приведённом упрощённом варианте Pow2 однозначно быстрее, хотя и крайне неэффективен по памяти.

**Вывод**

В данной лабораторной работе я рассмотрел два разных подхода к динамическому распределению памяти: алгоритм Мак–Кьюзи–Кэрелса и схему, основанную на блоках размера 2^n. Оба метода были реализованы в виде отдельных аллокаторов, с последующим проведением тестов на скорость выделения и освобождения памяти, а также на эффективность использования (учёт фрагментации и реального полезного объёма). По итогам экспериментов выявились яркие преимущества и недостатки каждого из способов.

Алгоритм Мак–Кьюзи–Кэрелса, опирающийся на списки свободных блоков и «best fit», позволяет достаточно тонко управлять памятью и эффективно подгонять выделения под нужные размеры, что повышает «фактор использования» и уменьшает внутреннюю фрагментацию. Однако при большом количестве обращений (особенно если в каждом случае нужно искать наиболее подходящий блок) время работы alloc заметно возрастает из-за линейного обхода списка и проверки всех кандидатов. С точки зрения управляемости и гибкости, Мак–Кьюзи–Кэрелс предоставляет возможности для оптимизации, например, можно ввести дополнительные таблицы/“segregated lists”, однако базовый вариант при интенсивной загрузке даёт довольно высокие накладные расходы.

Аллокатор, основанный на блоках 2^n, в простейшем исполнении фактически превращается в очень быстрый механизм, при котором все операции alloc и free проходят почти мгновенно: нужно лишь найти или вернуть блок заданного размера в соответствующий «карман» (список или массив списков). Однако из-за отсутствия тонкой подгонки под размер запроса итоговая фрагментация может быть великой, что приводит к низкому коэффициенту использования памяти. В более продвинутом варианте (buddy-аллокатор) множество системных компонентов и ядра операционных систем пользуются этим методом именно благодаря его высокой производительности и способности быстро делить и сливать блоки.

Таким образом, эксперименты подтвердили, что при упоре на скорость работы, когда важнее всего моментально выделять/освобождать память, логика «блоков по степени двойки» оказывается более подходящей. Если же приоритет смещён в сторону экономии памяти и снижения фрагментации, то «Мак–Кьюзи–Кэрелс» лучше справляется со своими задачами. Тесты также показали, что каждая из реализаций может быть существенно улучшена: алгоритм Мак–Кьюзи–Кэрелса можно ускорить (например, путём сегрегации списков по размерным классам), а схему 2^n можно доработать до полноценного buddy, обеспечив более рациональное объединение и разделение блоков. В реальной практике наиболее эффективными становятся гибридные решения, берущие лучшее из обоих подходов.