****Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Курсовая работа по курсу**

**«Операционные системы»**

**«Аллокаторы памяти»**

Группа: М80–206Б-18

Студент: Касимов М.М.

Преподаватель: Миронов Е.С.

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

Дата: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Москва, 2019.

**Содержание**

1. Постановка задачи
2. Общие сведения о программе
3. Аллокаторы памяти
4. Тестирование и обоснование подхода тестирования
5. Сравнение ключевых характеристик аллокаторов
6. Заключение
7. Список литературы

**Постановка задачи**

Необходимо реализовать два алгоритма аллокации памяти и сравнить их характеристики.

**Общие сведения о программе**

Программа была написана на языке С++ в среде Ubuntu 18.04.3. В программе используются STL контейнеры: list, map. Также используются библиотеки:

iostream (для ввода и ввывода), chrono (для измерения времени). Для отладки использовался gdb. Еще был написан простенький генератор тестов на языке C++.

Собирался проект утилитой cmake.

**Аллокаторы**

Во многих ОС важную роль играют так называемые менеджеры памяти т.е. аллокаторы. Они отвечают за управление физической памятью машины. Существуют разные алгоритмы, методы, на которых основываются аллокаторы. Я изучил два метода: метод близнецов и метод карты ресурсов с правилом наиболее подходящего участка.

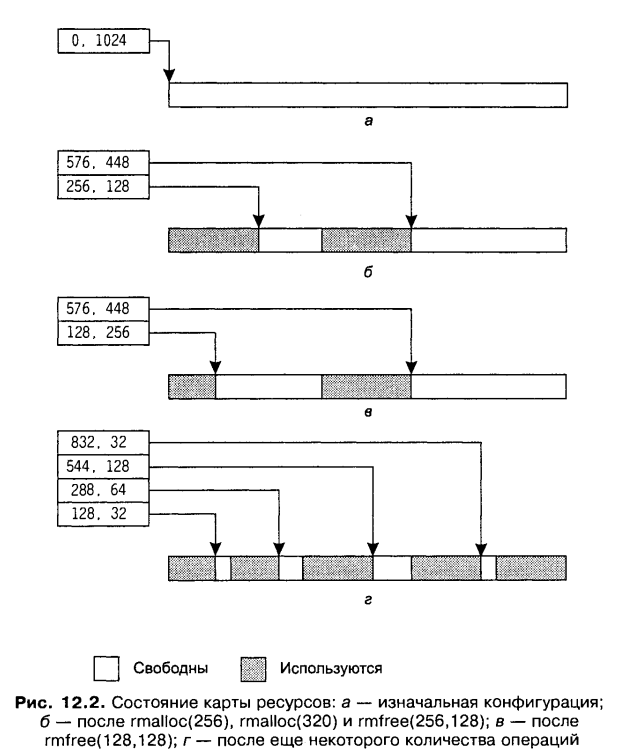
**Аллокатор карты ресурсов c правилом наиболее подходящего участка.**

Этот распределитель использует наборы пар <базовый адрес (смещение от начального адреса), размер>, благодаря которым он отслеживает свободные области памяти. Изначально область памяти описывается одной картой, в котором смещение равно 0, а размер равен ее общему объему памяти.

После этого клиенты начинают запрашивать и освобождать участки памяти, из-за чего область становится фрагментированной.

Если пользователю нужна память, то аллокатор, следуя правилу, проверяет все наборы карт ресурсов, с целью найти карту с наименьшим размером, который удовлетворит запрос. Когда такая область находится, то пользователю возвращается указатель на эту область.

Если пользователь хочет освободить память, которую он когда-то выделил, то аллокатор создает карту с данными о высвобождаемой области памяти и ищет куда его вставить в наборе карт. Упорядоченность набора карт важна, так как две соседние карты могут быть объединены в одну, большего размера, если они представляют две соседние области памяти. Это в какой-то доле решает проблему фрагментации, с которой у этого аллокатора итак все плохо. После вставки новой карты идет поиск карт, которые можно объединить. У данного аллокатора доступно частичное высвобождение из ранее выделенной памяти. В дальнейшем я буду пропускать в названии слова «с правилом наиболее подходящего участка».

[1].

Реализация этого аллокатора на С++:

#pragma once

#include <map>

#include <list>

#include <stdlib.h>

#include <iostream>

struct my\_pair {

size\_t base;

size\_t size;

};

struct Resource\_map\_allocator {

private:

std::list<my\_pair> l;

char \*memory\_pool;

size\_t size\_;

public:

Resource\_map\_allocator(size\_t n) {

size\_t m = n;

memory\_pool = (char\*) malloc(m);

l.push\_back({0, m});

size\_ = m;

}

void\* allocate(size\_t n) {

bool flag = false;

if (n < 1) {

return nullptr;

}

auto it = l.begin();

size\_t min = size\_;

for (auto i = l.begin(); i != l.end(); ++i) {

if (i->size >= n) {

if (i->size < min) {

it = i;

min = i->size;

}

flag = true;

}

}

if (flag) {

if ((it->size - n) != 0) {

l.push\_back({it->base + n, it->size - n});

}

size\_t tmp = it->base;

l.erase(it);

return (void \*) (memory\_pool + tmp);

} else {

return nullptr;

}

}

~Resource\_map\_allocator() {

free(memory\_pool);

}

template <class T>

int deallocate(T\* Ptr,size\_t size) {

bool flag = true;

char \*ptr = (char\*) Ptr;

size\_t tmp = ptr - memory\_pool;

for (auto i = l.begin(); i != l.end(); ++i) {

if ((i->base <= tmp) && ((i->base + i->size) > tmp)) {

return 1;

}

}

my\_pair new\_pair = {tmp,size};

for (auto i = l.begin(); i != l.end(); ++i) {

if (new\_pair.base < i->base) {

l.insert(i, new\_pair);

flag = false;

break;

}

}

if (flag) {

l.push\_back(new\_pair);

}

for (auto i = l.begin(); i != l.end(); ++i) {

auto tmp\_iter = i;

auto nxt = std::next(tmp\_iter);

if ((tmp\_iter->base + tmp\_iter->size) == nxt->base) {

my\_pair tmp\_pair = {tmp\_iter->base,tmp\_iter->size + nxt->size};

auto next\_next = std::next(i,2);

l.erase(tmp\_iter);

l.erase(nxt);

l.insert(next\_next,tmp\_pair);

break;

}

}

for (auto i = l.begin(); i != l.end(); ++i) {

auto tmp\_iter = i;

auto nxt = std::next(tmp\_iter);

if ((tmp\_iter->base + tmp\_iter->size) == nxt->base) {

my\_pair tmp\_pair = {tmp\_iter->base,tmp\_iter->size + nxt->size};

auto next\_next = std::next(i,2);

l.erase(tmp\_iter);

l.erase(nxt);

l.insert(next\_next,tmp\_pair);

break;

}

}

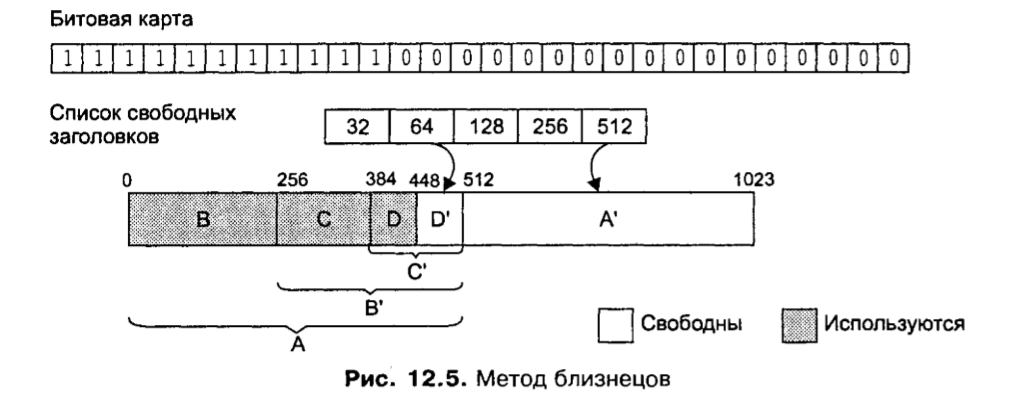
return 0;

}

};

**Аллокатор, основанный на методе близнецов**

Этот аллокатор сочетает в себе возможность слияния буферов и списки степени 2. В основе метода близнецов лежит создание буферов малого размера путем деления пополам больших буферов и слияния смежных буферов по мере возможности. Слияние соседних областей происходит если они оба свободные. При разделении буфера на два каждая половина называется близнецом второй.



В моей реализации размеры буферов были взяты с [1]. Хороший пример этого аллокатора продемонстрирован в [1].

Помимо списков степени 2, данный аллокатор использует так называемую битовую карту, которая говорит о состоянии порций блока (заняты они или свободны). Используя битовую карту, адрес и размеры буфера можно производить слияние буферов. В методе близнецов размер запрашиваемой памяти округляется до следующей степени числа 2 (т.е. для 17 байт отдаем 32).

Реализация на С++:

#pragma once

#include <map>

#include <list>

#include <stdlib.h>

#include <iostream>

const int MAX\_SIZE = 512;

const int MIN\_SIZE = 32;

const int TITLE2 = 5;

struct Buddy\_system\_allocator {

private:

char\* memory\_pool\_start;

char\* memory\_pool\_tail;

char\* memory\_pool\_end;

size\_t size\_;

std::list<char\*> title[TITLE2];

std::map<size\_t,size\_t> bitmap;

int32\_t power(int32\_t i) {

int pow = 0;

while(i > 1) {

i /= 2;

++pow;

}

return pow;

}

void rec\_deal(size\_t adrs,int index) {

bool flag = true;

size\_t temp = (adrs + (1 << index) \* 32);

if (!bitmap.count(temp) && temp < 1024) {

for (auto it = title[index].begin(); it != title[index].end(); ++it) {

if (\*it - memory\_pool\_start == (adrs + (1 << index) \* 32)) {

title[index].erase(it);

flag = false;

break;

}

}

if (flag) {

return;

}

title[index + 1].push\_back(memory\_pool\_start + adrs);

rec\_deal(adrs - (1 << (index + 1) \* 32),index + 1);

} else {

if (temp == 1024) {

title[index].push\_back(memory\_pool\_start + adrs);

}

return;

}

}

public:

Buddy\_system\_allocator(size\_t n) {

size\_t m = n \* 4096;

memory\_pool\_start = (char\*) malloc(m);

memory\_pool\_tail = memory\_pool\_start;

memory\_pool\_end = memory\_pool\_start + m;

size\_ = m;

}

~Buddy\_system\_allocator() {

free(memory\_pool\_start);

}

void\* allocate(size\_t n) {

bool flag = true;

if (n > MAX\_SIZE || n < 1) {

return nullptr;

}

uint32\_t index = 0;

uint32\_t buffer\_size = 32;

while (buffer\_size < n) {

++index;

buffer\_size <<= 1;

}

uint32\_t free\_index = index;

for (int i = index; i < TITLE2; ++i) {

if (!title[i].empty()) {

flag = false;

break;

}

++free\_index;

}

if (flag) {

if ((memory\_pool\_end - memory\_pool\_tail) < buffer\_size) {

return nullptr;

} else {

uint32\_t tmp = (1 << index) \* 32;

char\* start\_tmp = memory\_pool\_tail;

memory\_pool\_tail += 1024;

int32\_t diff = 1024;

while (diff != tmp) {

diff /= 2;

int32\_t array\_index = power(diff) - 5;

title[array\_index].push\_back(start\_tmp + diff);

}

bitmap[start\_tmp - memory\_pool\_start] = index;

return (void\*) start\_tmp;

}

} else {

auto it = title[free\_index].begin();

char\* tmp = \*it;

uint32\_t tmp\_diff = (1 << free\_index) \* 32;

title[free\_index].pop\_front();

if (index == free\_index) {

bitmap[tmp - memory\_pool\_start] = index;

return tmp;

} else {

while (index != free\_index) {

tmp\_diff /= 2;

--free\_index;

title[free\_index].push\_back(tmp + tmp\_diff);

}

bitmap[tmp - memory\_pool\_start] = index;

return tmp;

}

}

}

template <class T>

int deallocate(T\* Ptr) {

char\* ptr = (char\*) Ptr;

if (!bitmap.count(ptr - memory\_pool\_start)) {

return 1;

} else {

int a1 = ptr - memory\_pool\_start;

int a2 = bitmap[a1];

bitmap.erase(a1);

rec\_deal(a1,a2);

return 0;

}

}

};

**Тестирование и обоснование подхода тестирования**

Под выделением и освобождением будет подразумеваться аллокация и деаллокация соответственно.

Аллокатор карты ресурсов

Вход Результат

|  |  |  |
| --- | --- | --- |
| 1 | N = 1, т.е. выделение аллокаторам по 1 странице (4096 б.).  Выделение массива n1 char с количеством элем-в 2000.  Выделение массива n2 int с количеством элем-в 100.  Выделение массива n3 char с количеством элем-в 2000.  Освобождение массива n2.  Выделение массива n3 char с количеством элем-в 2000. | Memory for n1 is allocated  Memory for n2 is allocated  No memory allocated for n3  Memory for n3 is freed  Memory for n3 is allocated |
| 2 | N=1.  Выделение массива m1 char с количеством элем-в 512.  Выделение массива m2 char с количеством элем-в 512.  . . .  Выделение массива m8 char с количеством элем-в 512.  Освобождение массива m2.  Освобождение массива m4.  Освобождение массива m6.  Освобождение массива m8.  Выделение массива m9 char с количеством элем-в 513. | Memory for m1 is allocated  Memory for m2 is allocated  Memory for m3 is allocated  Memory for m4 is allocated  Memory for m5 is allocated  Memory for m6 is allocated  Memory for m7 is allocated  Memory for m8 is allocated  Memory for m2 is freed  Memory for m4 is freed  Memory for m6 is freed  Memory for m8 is freed  No memory allocated for m9 |
| 3 | N=1  Выделение массива k1 char с количеством элем-в 17.  Выделение массива k2 char с количеством элем-в 33.  Выделение массива k3 char с количеством элем-в 65.  Выделение массива k4 char с количеством элем-в 129. | Memory for k1 is allocated  17 bytes allocated  Memory for k2 is allocated  33 bytes allocated  Memory for k3 is allocated  65 bytes allocated  Memory for k4 is allocated  129 bytes allocated |
| 4 | Работа со списком. Добавление 100000 элементов и удаление,  снова добавление 100000 и удаление (объем элемента списка – 22 б.). | Выделение – 741542 мкс  Освобождение – 592498 мкс |

Аллокатор на алгоритме близнецов

Вход Результат

|  |  |  |
| --- | --- | --- |
| 1 | N = 1, т.е. выделение аллокаторам по 1 странице (4096 б.).  Выделение 16 массивов char с количеством элем-в 256.  Удаление трех последних массивов.  Выделение массива char с количеством элем-в 513. | Memory is allocated  Memory is allocated  Memory is allocated  Memory is allocated  Memory is allocated  Memory is allocated  Memory is allocated  Memory is allocated  Memory is allocated  Memory is allocated  Memory is allocated  Memory is allocated  Memory is allocated  Memory is allocated  Memory is allocated  Memory is allocated  Memory is freed  Memory is freed  No memory allocated |
| 2 | N=1.  Выделение 32 массивов char с количеством элем-в 128.  Освобождение второго, четвертого, шестого, . . . , 32-го массива.  Выделение массива char с количеством элем-в 129. | Memory is allocated  . . .  Memory is allocated  Memory is freed  . . .  Memory is freed  No memory allocated |
| 3 | N=1  Выделение массива k1 char с количеством элем-в 17.  Выделение массива k2 char с количеством элем-в 33.  Выделение массива k3 char с количеством элем-в 65.  Выделение массива k4 char с количеством элем-в 129. | Memory for k1 is allocated  32 bytes allocated  Memory for k2 is allocated  64 bytes allocated  Memory for k3 is allocated  128 bytes allocated  Memory for k4 is allocated  256 bytes allocated |
| 4 | Работа со списком. Добавление 100000 элементов и удаление,  снова добавление 100000 и удаление (объем элемента списка – 22 б.). | Выделение – 698257 мкс  Освобождение – 804107 мкс |

Первый тест показывает слияние буферов, которое позволяет уменьшить фрагментацию.

Второй тест показывает фрагментацию.

Третий – фактор использования.

Четвертый демонстрирует время. Тестировать аллокаторы на время очень трудно, так как приходится самому аккуратно заниматься выделением и освобождением памяти. Более, менее получилось поработать со списком, там хотя бы выделение и высвобождение происходит много раз, в то время как в моих попытках придумать что-то неординарное выводилось время 0 (из-за небольшого количества выделений).

Подробнее о каждом результате будет дальше.

**Сравнение ключевых характеристик аллокаторов**

Тесты и сравнение посвящены таким характеристикам аллокаторов: фактор использования, фрагментация, скорость выделения и высвобождения блоков. Первые две характеристики показывают эффективность т.е. экономность памяти при работе с ней. Это очень важно, ведь память машины конечна! Время работы аллокатора напрямую влияет на скорость работы ОС и других программ, которые используют динамически выделенную память. Это ключевые характеристики любого аллокатора, именно поэтому сравнивать я буду их.

У обоих аллокаторов имеется слияние соседних областей памяти, что является хорошим плюсом. Ведь если бы не было слияния, то две соседние области представляли бы два фрагмента, и если бы пользователь хотел запросить память, которая больше обеих областей по отдельности, но меньше их совокупности, то он не получил бы памяти на запрос. Явным отличием аллокатора, который использует метод близнецов, является ограниченность выделяемой пользователю памяти (в [1] это 512 байт). Т.е. если в нашем аллокаторе на алгоритме близнецов максимальный размер среди списков степени два – 512 и при этом есть гораздо больше свободной памяти (имеется виду больше 512), то данный аллокатор не сможет удовлетворить запрос, скажем на 513 байтов. В то время как аллокатор карты ресурсов может удовлетворить запрос любого объема, если есть на это свободная память.

Как видим из второго теста, у обоих аллокаторов присутствует довольна заметная фрагментация. Это очень плохо, так как мы иногда не сможем удовлетворить запрос пользователя, даже когда свободной памяти больше половины от общей, из-за большого количества фрагментов, которые по отдельности меньше требуемой памяти. Так как алгоритм близнецов в основном поддерживает освобождение только целого буфера (частичное освобождение является неэффективным, т.к. неполный буфер не будет иметь близнеца и в дальнейшем это плохо скажется на алгоритм. Смысл алгоритма в близнецах, благодаря которым можно рекурсивно производить слияние соседних областей т.е. близнецов!), он менее подвержен фрагментации, что не скажешь о карте ресурсов. Ведь в аллокаторе карты ресурсов возможно частичное высвобождение, что увеличивает количество фрагментов.

Фактор использования у аллокатора карты ресурсов намного лучше, чем у аллокатора близнецов. Если имеется память, удовлетворяющая запросу, то этот аллокатор выдаст ее со 100% фактором использования. В то время как аллокатор близнецов на запрос 17 байтов использует (т.е. отдает) 32. В этом случае фактор использования равен 53%. У данного аллокатора фактор использования держится на уровне 50%, так как входное значение округляется кверху, к степени 2 (например, 133 к 256) и такое отношение меньшего к большему не может быть меньше ½. Стоит отметить, что в аллокаторе на списках степени 2 фактор использования гораздо меньше, т.к. там буферы не способны менять свой размер, в то время как в методе близнецов буферы делятся пополам до ближайшего объема.

Оба аллокатора являются низкопроизводительными по времени. Карте ресурсов приходится искать место для карты, для поддержания упорядоченности набора карт, приходится часто выполнять операцию последовательного поиска в карте с целью обнаружения достаточно большого для удовлетворения запроса участка, но при этом минимального в наборе. Уходит время и на поиск карт, которые могут быть объединены. Так же при деаллокации нужно проверять набор карт, чтобы не «высвободить» уже свободный участок памяти. У аллокатора близнецов тоже все плохо, он тратит много времени на операцию слияния буферов, которая к тому же рекурсивна. Именно поэтому в 4 тесте (там слияние происходит 2\*n раз, где n количество буферов, так как мы удаляем последовательно соседние буферы и удаление всех элементов происходит 2 раза) он показал такой плохой результат скорости высвобождений.

**Заключение**

Таким образом, методика близнецов обладает такими хорошими свойствами: соединение смежных блоков, возможность изменения размеров участков памяти и их повторного использования в преобразованном виде (т.е. деление пополам), не самый лучший, но довольно хороший фактор использования в виде 50%. Из минусов: низкая производительность.

В свою очередь карта ресурсов с правилом наиболее подходящего участка имеет такие плюсы: простота алгоритма для реализации, отличный фактор использования в 100%, соединение двух последовательных участков памяти в один. Из минусов данного метода: сильная фрагментация, которую увеличивает частичное высвобождение, очень низкая производительность.

При выполнении данной курсовой работы я исследовал два вида аллокаторов памяти, познакомился с новыми для себя понятиями, такими как фактор использования. Я сравнил эти аллокаторы, благодаря этому я понял, какой аллокатор с чем справляется хорошо, а с чем плохо. Данная курсовая работа, поспособствовала к изучению UNIX системы, помогла мне применить практические навыки, полученные на курсах операционных систем и ООП на С++.

**Список литературы**

1.Вахалия Ю. «UNIX изнутри».