Московский Авиационный Институт
(Национальный исследовательский университет)

Курсовой проект
По курсу:
«Операционные системы»

Студент: Забарин Н.И.
Группа: 8О – 408Б

Москва, 2016

Цель курсового проекта

Исследование характеристик алгоритмов аллокации памяти.

Задание

Необходимо реализовать два алгоритма аллокации памяти и сравнить их по следующим параметрам:

- 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.

В отчете необходимо отобразить следующее:

- Подробное описание каждого из исследуемых алгоритмов
- Процесс тестирования
- Обоснование подхода тестирования
- Результаты тестирования
- Заключение по проведенной работе

Вариант

Я выбрал для сравнения следующие алгоритмы:

- Выбор первого подходящего (далее ПП-алгоритм)
- Выбор лучшего подходящего (далее ЛП-алгоритм)

Реализация на языке C++11

```
#include "stdlib.h"
#include "stdint.h"
#include "stdio.h"
#include "string.h"
#include <ctime>
#include <iostream>
#include <vector>

//#define SIMPLE

//#define LOGS

using namespace std;

typedef struct element_s {
    struct element_s *next, *prev;
    bool used;
    size_t size;
} element_t;

#define FOREACH(I, HEAD) for(I=HEAD;I!=NULL&&I->next!=I;I=I->next)
#define FOREACH_SAFE(I, HEAD, TMP) for(I=HEAD,TMP=I->next;(I!=NULL)and(TMP=I->next);I=TMP)

#define INSERT(NEW, ELEM)          \
{                                  \
    if (ELEM->next != NULL) {      \
        ELEM->next->prev = NEW;    \
    }                              \
    NEW->next = ELEM->next;        \
```

```
    ELEM->next = NEW;               \
    NEW->prev = ELEM;               \
}

#define DELETE(ELEM)                    \
{                                   \
    if (ELEM->next != NULL) {           \
        ELEM->next->prev = ELEM->prev;    \
    }                               \
    if (ELEM->prev != NULL) {           \
        ELEM->prev->next = ELEM->next;    \
    }                               \
}

#define MERGE_NEXT(ELEM)                                        \
{                                                   \
    if (ELEM!=NULL){                                    \
        if (ELEM->next!=NULL and !(ELEM->used) and !(ELEM->next->used)) {  \
            ELEM->size += ELEM->next->size + sizeof(element_t);           \
            DELETE(ELEM->next);                                 \
        }                                               \
    }                                               \
}

#define SUM(L, K, TYPE) (TYPE)((uint64_t)L + (uint64_t)K)
#define SUB(L, K, TYPE) (TYPE)((uint64_t)L - (uint64_t)K)


element_t *head = NULL;
size_t allocated = 0;

void init(size_t size) {
    if (size <= sizeof(element_t)) {
        cout << "ERROR: Too small size." << endl;
    }
    head = (element_t *)calloc(size, 1);
    head->used = false;
    head->size = size - sizeof(element_t);
    head->next = NULL;
    head->prev = NULL;
#ifdef LOGS
    cout << "Initialization: " << size << " bytes" << endl;
#endif
    allocated = size;
}

void *alloc(size_t size) {
    element_t *elem = NULL, *nelem = NULL;
    element_t *res = NULL;
#ifdef SIMPLE
    FOREACH(elem, head) {
        if (!elem->used and elem->size >= size) {
            size_t excess_mem = elem->size - size - sizeof(element_t);
#ifdef LOGS
            cout << "Allocation:   " << size << " bytes" << endl;
#endif
            if (elem->size > size + sizeof(element_t)) {
                res = SUM(elem, sizeof(element_t), element_t*);
                nelem = SUM(res, size, element_t*);
                nelem->used = false;
                nelem->size = excess_mem;
                INSERT(nelem, elem);
                elem->used = true;
                elem->size = size;
                return res;
            } else {
                res = SUM(elem, sizeof(element_t), element_t*);
                elem->used = true;
                return res;
            }
```

```
        }
    }
#else
    element_t *best = NULL;
    FOREACH(elem, head) {
        if (!elem->used and elem->size >= size) {
            if (best == NULL or best->size > elem->size){
                best = elem;
                //cout << "1";
            }
        }
    }

    if (best != NULL) {
        //cout << best->size - size << endl;
        size_t excess_mem = best->size - size - sizeof(element_t);
#ifdef LOGS
        cout << "Allocation:   " << size << " bytes" << endl;
#endif
        if (best->size > size + sizeof(element_t)) {
            res = SUM(best, sizeof(element_t), element_t*);
            nelem = SUM(res, size, element_t*);
            nelem->used = false;
            nelem->size = excess_mem;
            INSERT(nelem, best);
            best->used = true;
            best->size = size;
            return res;
        } else {
            res = SUM(best, sizeof(element_t), element_t*);
            best->used = true;
            return res;
        }
    }
#endif

#ifdef LOGS
    cout << "Allocation:   " << size << " bytes. FAIL" << endl;
#endif
    return NULL;
}

void dealloc(void *l) {
    element_t *elem = SUB(l, sizeof(element_t), element_t*);
    elem->used = false;
#ifdef LOGS
    cout << "Deallocation: " << elem->size << " bytes" << endl;
#endif
    MERGE_NEXT(elem);
    MERGE_NEXT(elem->prev);
}

void print_list() {
    element_t *elem = NULL;
    cout << "List of blocks:" << endl;
    int i = 0;
    FOREACH(elem, head) {
        printf("  %d: used %u, size of block %lu bytes", i, elem->used, elem->size);
        // DEBUG print pointers
        //printf(" link %p next %p prev %p\n", elem, elem->next, elem->prev);
        cout << endl;
        i++;
    }
}

#define BUF_SIZE 150

void print_mem_usage() {
    char buf[BUF_SIZE+1];
    memset(buf, '.', BUF_SIZE);
```

```
    buf[BUF_SIZE] = '\0';

    element_t *elem = NULL;
    size_t prev = 0;
    size_t used_mem = 0;
    size_t sys_mem = 0;
    size_t _mem = 0;

    FOREACH(elem, head) {
        int p = prev * BUF_SIZE / allocated;
        if (elem->used) {
            _mem += elem->size + sizeof(element_t);
            used_mem += elem->size;
        } else {
            int n = _mem * BUF_SIZE / allocated;
            memset(buf+p, '|', n);
            prev += _mem + sizeof(element_t) + elem->size;
            _mem = 0;
        }
        used_mem += sizeof(element_t);
        sys_mem += sizeof(element_t);
    }
    float per_u = (float)(used_mem * 100) / allocated;
    float per_s = (float)(sys_mem * 100) / allocated;

    cout << "Memory usage: " << endl;
    cout << '\t' << '[' << buf << ']' << endl;
    printf("\t%.2f", per_u);
    cout << "\% of memory used." << endl;
    printf("\t%.2f", per_s);
    cout << "\% of memory used by allocator." << endl;
}

void destroy() {
    element_t *elem = NULL;
    FOREACH(elem, head) {
        elem->used = false;
    }
#ifdef LOGS
    cout << "Destroying" << endl;
#endif
    free((void *)head);
}

double *test() {
   double *res = (double*)calloc(6, sizeof(double));
#define ITER_NUMBER 10
   for (int j = 0; j < ITER_NUMBER; ++j) {
        char fin[10];
        snprintf(fin, 10, "gtest%d", j);
        freopen(fin, "r", stdin);
        size_t total, sum = 0;
        int n, real_n, a_cnt = 0, d_cnt = 0, f_cnt = 0;
        cin >> total >> n;
        real_n = n;
        init(total);
        vector<void*> elements;
        clock_t a_time = 0, d_time = 0, tmp;
        for (int i = 0; i < n; ++i) {
            int action, v;
            cin >> action >> v;
            if (action == 0) { // 0 - alloc
                tmp = clock();
                void *new_elem = alloc(v);
                a_time += clock() - tmp;
                ++a_cnt;
                sum += v;
                if (new_elem != NULL) {
                    elements.push_back(new_elem);
                    memset(new_elem, 0, v);
```

```
            } else {       // allocation failed
//                cout << "FAIL\n";
                f_cnt += 1;
            }
        } else if (action == 1 && elements.size() > 0) { // 1 - dealloc
            //v = v % elements.size();
            tmp = clock();
            dealloc(elements[v]);
            d_time += clock() - tmp;
            ++d_cnt;
            elements.erase(elements.begin() + v);
        } else if (action == 2) {
            print_mem_usage();
            printf("\n");
            --real_n;
        }
      }
      res[0] += a_cnt;
      res[1] += d_cnt;
      res[2] += a_time;
      res[3] += d_time;
      res[4] += sum;
      res[5] += f_cnt;
    //cout << "Total operations " << real_n << ", allocations " << a_cnt << ", deallocations " << d_cnt << endl;
    //if (a_cnt == 0) ++a_cnt;
    //if (d_cnt == 0) ++d_cnt;
    //cout << "Avegare size of element " << (uint64_t)(sum / a_cnt) << endl;
    //cout << "Avegare allocation time " << ((float)a_time / CLOCKS_PER_SEC / a_cnt) << endl;
    //cout << "Avegare deallocation time " << ((float)d_time / CLOCKS_PER_SEC / d_cnt) << endl;
      fclose(stdin);
      destroy();
      //++(fin[5]);
  }
  for (int i = 0; i < 6; ++i) res[i] /= ITER_NUMBER;
  return res;
}


int main() {
 cout << sizeof(element_t) << endl;
 return 0;
  srand (time(NULL));
#ifdef SIMPLE
  cout << "Simple allocation" << endl;
#else
  cout << "Best allocation" << endl;
#endif
  //test2(10*1000, 1000, 500, 500);
  //test2(10*1000, 1000, 1000, 100);
  double *res = test();

  cout << "Allocations " << res[0] << ", deallocations " << res[1] << endl;
  if (res[0] == 0) ++res[0];
  if (res[1] == 0) ++res[1];
  cout << "Avegare size of element " << (uint64_t)(res[4] / res[0]) << endl;
  cout << "Avegare allocation time " << ((float)res[2] / CLOCKS_PER_SEC / res[0]) << endl;
  cout << "Avegare deallocation time " << ((float)res[3] / CLOCKS_PER_SEC / res[1]) << endl;
  cout << "Average fails counter " << res[5] << endl;
}
```

Описание реализации

Данные о блоках хранятся в двунаправленном списке. Эта структура позволяет вставлять и удалять элементы за O(1), очевидным минусом является поиск за линейное время. Структура элемента списка:

```
typedef struct element_s {
  struct element_s *next, *prev;
  bool used;
  size_t size;
} element_t;
```

Структура довольно проста, ссылки на предыдущий и следующий элементы, размер блока и флаг, говорящий о том используется блок или нет.

Реализованы несколько функций-дефайнов для работы со списками, код дефайнов был взят из *этого проекта*, и немного переписаны под мои нужды. Среди них:

- *FOREACH* — пробежка по всему списку
- *FOREACH_SAFE* — пробежка по всему списку с сохранением след. элемента, для безопасного удаления/изменения текущего элемента
- *INSERT* — вставка нового элемента
- *DELETE* — удаление элемента
- *MERGE_NEXT* — попытка слияния элемента со следующим, это моя функция.

Два дефайна для корректных сложения и вычитания ссылок.

Далее идут функции для работы с моим аллокатором:

- *init(size)* — выделяет стандартными средствами size байт памяти и создает первый блок. Эту функцию необходимо вызывать для работы аллокатора.
- *alloc(size)* — возвращает ссылку на блок памяти размером size или NULL, если память выделить не удалось.
- *dealloc(link)* — освобождает блок памяти выделенный с помощью alloc.
- *destroy()* — освобождает память выделенную под аллокатор.

А так же несколько вспомогательных(отладочных) функций:

- *print_list()* — выводит все блоки памяти.
- *print_mem_usage()* — выводит текущее состояние памяти, немного криво отображается, при большом кол-ве маленьких блоков. Так же подсчитывает процент памяти занимаемый самим аллокатором, то есть элементами списка.

Используемые алгоритмы

Сами по себе ПП и ЛП -алгоритмы довольно тривиальны. Оба представляют собой пробег по списку, ПП останавливается, когда находит незанятый элемент размером больше, чем необходимо, ЛП же всегда пробегает весь список элементов и пытается найти наименьший элемент подходящего размера. В этом и состоит разница в алгоритмах, когда списки набирают большую длину оба алгоритма работают медленно, но ПП всегда не медленнее, чем ЛП. Преимуществом ЛП алгоритма является, то что он более рационально распределяет блоки памяти, что я показал на примере 3 теста, описанного ниже.

Методика сравнения алгоритмов выделения памяти

Что бы сравнить алгоритмы я написал генератор случайных тестов. У каждого теста есть несколько параметров: максимальный объем выделяемой памяти, число запросов, минимальный и максимальный размер выделяемого блока, а так же отношение числа аллокаций к числу освобождений.

За раз генерируется 10 файлов-тестов, потом программа последовательно выполняет команды из файлов, посчитывая среднее время выполнения аллокаций и освобождений, а так же средний размер выделяемого блока и количество ошибок при выделении — алгоритм не нашел подходящего блока памяти. Сразу после выделения блока весь его объем заполняется нулями для проверки корректности работы программы. После выполнения всех 10 тестов подсчитываются средние значения указанных выше параметров.

На основании этих усредненных результатов я и буду сравнивать алгоритмы.

Формат ввода теста

Файл-тест выглядит следующим образом:

<Объем памяти для инициализации программы>
<Кол-во операций>
<Идентификатор операции> <Параметр операции>
<Идентификатор операции> <Параметр операции>
<Идентификатор операции> <Параметр операции>
…

<Идентификатор операции> - число от 0 до 2. 0 — выделить блок памяти размером с *<Параметр операции>*. 1 — освободить блок памяти с номером *<Параметр операции>*. 2 — вывести состояние памяти.

Пример:

10000000
10001
0 6558
1 0
0 9700
0 7149
1 0
0 1040
0 1149
0 5946
0 8771
1 4
1 2
….

Результат работы программы

Тест 1.

Best allocation
Memory usage:
[.....|..|.....|.....|....|.....|...|.||.||..]
96.88% of memory used.
0.76% of memory used by allocator.
Allocations 5998.3, deallocations 4001.7
Avegare size of element 5516
Avegare allocation time 1.63094e-05
Avegare deallocation time 2.04863e-07
Average fails counter 182.5

Simple allocation
Memory usage:
[.....|.....|.]
93.21% of memory used.
0.79% of memory used by allocator.

Allocations 5998.3, deallocations 4001.7
Avegare size of element 5516
Avegare allocation time 1.10986e-05
Avegare deallocation time 2.43147e-07
Average fails counter 227.5

В данном тесте выделялось 10Mb памяти, 10 000 запросов из которых 60% на аллокацию. Размер выделяемого блока от 1Kb до 10Kb.

На примере этого довольно простого теста можно увидеть, что ЛП-алгоритм выделяет память на 47% медленнее в среднем, но при этом ошибся на 45 раз меньше. Поскольку тесты генерируются случайным образом, для части запросов на выделение может не хватать памяти независимо какой алгоритм выделяется.

Тест 2.

Best allocation
Memory usage:

[.....]
95.74% of memory used.
0.01% of memory used by allocator.

Allocations 66675.7, deallocations 33324.3
Avegare size of element 549995
Avegare allocation time 0.000133957
Avegare deallocation time 3.94634e-07
Average fails counter 30497

Simple allocation
Memory usage:
[.....]
91.45% of memory used.
0.01% of memory used by allocator.

Allocations 66675.7, deallocations 33324.3
Avegare size of element 549995
Avegare allocation time 0.00010727
Avegare deallocation time 4.19682e-07
Average fails counter 30677.2

В данном тесте выделялось 1Gb памяти, 100 000 запросов из которых 66% на аллокацию. Размер выделяемого блока от 100Kb до 1Mb.

При большом кол-ве запросов разница в количестве ошибок почти незаметна, а вот проигрыш в скорости ЛП-алгоритмом все еще составляет порядка 34%. Зато видно, что в конце работы программы ЛП-алгоритм занимает почти 96% памяти, а ПП всего 91,5%, то есть разница в выделенной памяти составляет 45Mb.

Тест 3.

Не рандомный тест. Выделяесть 10 Мб памяти, далее она вся выделяется блоками размером 1-2Кб, потом освобождаем каждый второй блок, запоминая размер удаляемых блоков. И снова их выделяем.

Best allocation
Allocations 9738.3, deallocations 3246.3
Avegare size of element 1500
Avegare allocation time 6.62108e-05
Avegare deallocation time 1.86828e-07
Average fails counter 0

Simple allocation
Allocations 9738.3, deallocations 3246.3
Avegare size of element 1500
Avegare allocation time 5.04132e-05
Avegare deallocation time 1.92927e-07
Average fails counter 128.7

На примере этого теста можно увидеть, что в некоторых ситуациях в которых память выделить можно, ПП-алгоритм ошибается, а ЛП-алгоритм нет, так же можно заметить что процент ошибок ПП-алгоритма достаточно мал и составляет всего 4%, тогда как проигрыш по скорости остается все теми же 32%.

Выводы или зачем это нужно

Стандартные средства выделения памяти (malloc/calloc/realloc/new и тд) унифицированы для любых нужд, и одинаково действуют, когда нужно выделить память под пару интов и под большой массив данных. Так же одним из минусов стандартных аллокаторов является то, что каждый его вызов это запрос к ядру системы, то есть при большом количестве вызовов это сильно замедляет работу системы, поэтому в серьезных проектах часто используются нестандартные методы выделения памяти, самописные или написанные кем-то до вас. Выбор же алгоритма сильно зависит от использования.

Например для хранения деревьев удобно использовать кольцевой алгоритм выделения памяти, когда сразу выделяется большой массив памяти разделенный на блоки одинакового

размера(размер равен памяти занимаемой одной вершиной дерева), ссылки на свободные блоки хранятся в кольце, для получения блока просто берется ссылка из кольца и удаляется оттуда, очищение это просто возврат ссылки обратно в кольцо. У этого метода есть ряд преимуществ:

- запросы в ядро только когда необходимо увеличить число блоков
- поскольку все вершины дерева лежат в памяти «подряд» это заметно уменьшает количество промахов в кэш
- если память выделяется с SSD диска, то кольцо дает какую-то равномерность использования диска, то есть продлевает его срок жизни

Если же говорить об алгоритмах представленных выше, то оба алгоритма используются редко, в силу их медлительности при больших количествах блоков. Но если выбирать из них, то ПП я бы использовал, когда задача подразумевает редкие освобождения памяти или когда все блоки имеют +/- одинаковый размер, а ЛП, когда необходимо максимально экономить память, и задача подразумевает блоки различных размеров и частое их освобождение.