**ЛАБОРАТОРНА РОБОТА #2**

З дисципліни «Системне програмне забезпечення»

На тему «Алокатор пам’яті загального призначення 2»

|  |  |
| --- | --- |
| **Виконав**  Студент групи КВ-21  Комарницький О.Б. | **Перевірила**  Ст. викл. кафедри СПіСКС  Дробязко І.П. |

|  |  |  |
| --- | --- | --- |
| **Оцінка** | **Дата** | **Підпис** |
|  |  |  |

**Завдання:** (Таке ж як у першій лабораторній роботі)

Розробити алокатор загального призначення, використовуючи за основу описаний базовий варіант алгоритму, враховуючи такі умови:

1. Області пам’яті можна виділяти будь-яким доступним способом.
2. Функції mem\_alloc(), mem\_realloc(), mem\_free() повинні відповідати наведеним прототипам.
3. Адрес пам'яті, що повертається функціями mem\_alloc(), mem\_realloc() мають бути вирівняними на межу 4 байтів.
4. Постаратись зменшити час пошуку вільного блоку пам’яті і час вивільнення зайнятого блоку.
5. Постаратися зменшити фрагментацію пам’яті.
6. Написати функцію mem\_dump(), що виводить в консоль стан областей пам’яті.

Звіт має містити:

1. Опис розробленого алгоритму.
2. Оцінку часу пошуку вільного блоку пам’яті, оцінку часу вивільнення зайнятого блоку.
3. Оцінку витрати пам’яті для збереження службової інформації.
4. Опис переваг і недоліків розробленого алокатора.
5. Лістинг алокатора пам’яті загального призначення.
6. Приклад роботи алокатора.
7. **Опис розробленого алгоритму.**

Розроблений алгоритм алокатора, загалом, працює згідно із методичними вказівками, а також враховує такі аспекти:

* Пошук вільної пам’яті здійснюється із молодших адрес.
* При виділенні пам’яті на розмір більший половини сторінки спочатку здійснюється пошук вільних блоків потрібного розміру у поділених сторінках.
* Виділити пам’ять можна як на декілька сторінок, так і на декілька блоків.
* Після виклику mem\_free() перевіряються блоки у сторінці, якщо всі вільні – то відбувається надання вільного статусу сторінці, що дає можливість знову розподілити її на блоки іншого розміру.
* Можна звільнити сторінку цілком, якщо вона зайнята багатосторінковим блоком.
* mem\_realloc() коли необхідно розширити блок, спочатку перевіряється можливість використати сусідні блоки, якщо їх пам’яті недостатньо, то вміст блоку цілком переноситься у блок більшого розміру.
* Алокатор може змінити як розмір вмісту сторінки, так і розмір вмісту блоку.
* Якщо необхідно зменшити розмір сторінки – то розглядається варіант створення блоків ефективного розміру, куди можна помістити вміст нового заданого розміру.
* Алокатор може збільшити вміст розміру блоку пошуком вільного місця пам’яті на величину збільшеного нового розміру вмісту блоку.

**2. Оцінка часу вивільнення та виділення пам'яті.**

Оцінка часу пошуку вільного блоку пам’яті:

З метою зменшення фрагментації пам’яті пошук вільного блоку ведеться із молодших адрес.

Оскільки спочатку здійснюється пошук поділених сторінок з метою зменшення фрагментації пам’яті, тому кількість сторінок і блоків у пам’яті суттєво вплинуть на швидкодію алокатора.

Потім здійснюється пошук вільних блоків групи 2, час якого залежить тільки від кількості сторінок.

Найдовше відбуватиметься створення блоків групи 1 у вільній сторінці. Така послідовність вибрана невипадково, адже значно більша імовірність того, що доведеться шукати вільні блоки у поділених сторінках великого розміру.

Оцінка часу вивільнення зайнятого блоку (сторінки):

Оскільки пошук блоку здійснюється з початку адрес, тому час пошуку буде залежати від кількості сторінок, розміщених в молодших адресах, а також буде залежати від кількості блоків у сторінці. Якщо блок чи сторінка не знайдені в даному діапазоні адрес, то функція завершується.

Крім того, після кожного виклику функції вивільнення зайнятих блоку чи сторінки здійснюється перевірка на можливість вивільнення сторінки цілком. Перевірка завершується знаходженням першого зайнятого блоку або ж коли всі блоки проаналізовані.

**3. Оцінка витрати пам’яті для збереження службової інформації.**

* Для блоків потреба у збереженні службової інформації відпадає, адже все, що необхідно для адресації і проходження по блоках можна знайти у дескрипторі сторінки. Якщо всі блоки однакового розміру, то проходитися по ним можна знаючи розміри блоку та розмір сторінки.

Якщо ж блок багатосторінковий, то тоді необхідно знати лише статус сторінок, і знову ж таки, ця інформація зберігається у дескрипторах.

* Службова інформація сторінок зберігається у дескрипторах, які відділені від блоків з користувацькою інформацією .

**4. Опис переваг і недоліків розробленого алокатора.**

Переваги:

1. При виділенні пам’яті пошук блока розпочинається із молодших адрес, що покращує систематизованість блоків пам’яті.
2. При вивільненні пам’яті здійснюється перевірка на можливість звільнення сторінки цілком.
3. Алокатор спершу шукає вільні блоки, а вже потім вільні сторінки чи пробує поділити сторінку на блоки, що зменшує фрагментованість.
4. Алокатор може змінити розмір як вмісту сторінки, так і блоку.

Недоліки:

1. При виділенні пам’яті необхідно пройти всі існуючі блоки.
2. Додаткові операції при вивільненні пам’яті щодо вивільнення сторіки.
3. Багато перевірок при зміненні розміру блоку, що зумовлює складність алокатора.

**5. Лістинг алокатора загального призначення.**

***Файл Functions.h***

enum State {FREE, BUSY, PARTED};

struct Descriptor

{

int free\_blocks\_count;

size\_t first\_free\_block;

size\_t addr, block\_size;

State state;

};

struct Block

{

size\_t addr, size;

State state;

};

struct Settings

{

size\_t page\_size = 4096;

int pages\_number = 10;

};

void \* mem\_alloc(size\_t size);

void \* mem\_realloc(void \* addr, size\_t size);

void mem\_free(void \* addr);

void mem\_dump();

void previous\_load();

***Файл Functions.cpp***

void \* mem\_alloc(size\_t size)

{

cast\_to4(size);

// Find place

if (size)

{

// At first find already parted pages

for (auto & d : Descriptors)

{

if (d.state == PARTED && (int)get\_space\_addr(size, d.addr) >= 0)

{

auto page = Memory.begin();

advance(page, get\_page\_number(d.addr));

size\_t free\_addr = get\_space\_addr(size, d.addr);

int N = get\_block\_number(d.addr, free\_addr);

Hash\_table[free\_addr] = size;

list<Block>::iterator b = (\*page).begin();

advance(b, N);

for (; size; (\*b).state = BUSY, b++)

size >= (\*b).size ? size -= (\*b).size : size = 0;

return (size\_t \*)free\_addr;

}

}

// Group 2. Parted pages not found, look for fully free pages.

if (size > settings.page\_size / 2)

{

int addr = get\_space\_addr(size);

if (addr >= 0)

{

list<Descriptor>::iterator page = Descriptors.begin();

advance(page, get\_page\_number((size\_t)addr));

Hash\_table[(\*page).addr] = size;

for (; size; (\*page).state = BUSY, page++)

size >= settings.page\_size ? size -= settings.page\_size : size = 0;

return (size\_t \*)addr;

}

return NULL;

}

// Same for group 1

for (auto & d : Descriptors)

{

if (d.state == FREE)

{

// Create blocks in page

auto page = Memory.begin();

advance(page, get\_page\_number(d.addr));

cast\_to16(size);

for (int i = 0; i < (int)(settings.page\_size / size); ++i)

{

Block b;

b.addr = d.addr + i \* size;

b.size = size;

b.state = FREE;

(\*page).push\_back(b);

}

(\*(\*page).begin()).state = BUSY;

Hash\_table[d.addr] = size;

d.block\_size = size;

d.first\_free\_block = -1; //ToDo

d.state = PARTED;

d.first\_free\_block = d.addr + settings.page\_size + size;

return (size\_t \*)d.addr;

}

}

}

return NULL;

}

void \* mem\_realloc(void \* addr, size\_t size)

{

// Find page

auto page = Memory.begin();

int p\_num = get\_page\_number((size\_t)addr);

if (p\_num < 0) return NULL; // Wrong page address

advance(page, p\_num);

auto desc = Descriptors.begin();

advance(desc, p\_num);

cast\_to4(size);

// If pages to reallocate

if (!(\*page).size() && Hash\_table.find((size\_t)addr) != Hash\_table.end())

{

if (size == 0)

{

int pages\_to\_realloc = (int)(ceil)((double)Hash\_table[(size\_t)addr] / settings.page\_size);

while (pages\_to\_realloc--)

{

(\*desc).state = FREE;

desc++;

}

return NULL;

}

else if (size < settings.page\_size) // Decrease size

{

int pages\_to\_realloc = (int)(ceil)((double)Hash\_table[(size\_t)addr] / settings.page\_size);

auto tmp = desc;

while (pages\_to\_realloc--) (\*tmp++).state = FREE;

size\_t realloc\_size = settings.page\_size - size;

size\_t b\_size = size;

// Choosing new blocks size

if (fmin(settings.page\_size - size, size) > settings.page\_size / 4)

b\_size = settings.page\_size / 4;

else // Blocks must be 2^n

cast\_to16(b\_size);

// Adding blocks with calculated size

Hash\_table[(size\_t)addr] = size;

for (size\_t i = 0; i < (int)(settings.page\_size / b\_size); i++)

{

Block b;

b.addr = (\*desc).addr + i \* b\_size;

b.size = b\_size;

b.state = size ? BUSY : FREE;

(\*page).push\_back(b);

size = ((int)size - (int)b\_size > 0) ? size - b\_size : 0;

}

(\*desc).state = PARTED;

return (size\_t \*)addr;

}

else if (size < Hash\_table[(size\_t)addr])

{

int new\_pages\_count = (int)(ceil)((double)size / settings.page\_size);

int pages\_to\_release = (int)(ceil)((double)Hash\_table[(size\_t)addr] / settings.page\_size) - new\_pages\_count;

while (new\_pages\_count--) desc++;

while (pages\_to\_release--) (\*desc++).state = FREE;

return (size\_t \*)addr;

}

else if (size > Hash\_table[(size\_t)addr])

{

int pages\_to\_release = (int)(ceil)((double)Hash\_table[(size\_t)addr] / settings.page\_size);

while (pages\_to\_release--) (\*desc++).state = FREE;

return mem\_alloc(size);

}

}

// If blocks to reallocate

else if ((\*page).size() && Hash\_table.find((size\_t)addr) != Hash\_table.end())

{

auto b = (\*page).begin();

advance(b, get\_block\_number((\*desc).addr, (size\_t)addr));

if (size == 0) // Destroy block

{

int blocks\_to\_realloc = (int)(ceil)((double)Hash\_table[(size\_t)addr] / (\*b).size);

while (blocks\_to\_realloc--) (\*b++).state = FREE;

// If all blocks are free

for (auto & block : (\*page))

if (block.state != FREE)

return (size\_t \*)(\*b).addr;

(\*desc).state = FREE;

(\*page).clear();

Hash\_table.erase((size\_t)addr);

return NULL;

}

else if (size >= (\*b).size && size <= Hash\_table[(size\_t)addr])

{

int new\_blocks\_count = (int)(ceil)((double)size / (\*b).size);

int blocks\_to\_release = (int)(ceil)((double)Hash\_table[(size\_t)addr] / (\*b).size) - new\_blocks\_count;

while (new\_blocks\_count--) b++;

while (blocks\_to\_release--) (\*b++).state = FREE;

Hash\_table[(size\_t)addr] = size;

return (size\_t \*)addr;

}

else if (size > Hash\_table[(size\_t)addr]) // Increase block's content space

return mem\_alloc((size\_t)size - Hash\_table[(size\_t)addr]);

}

return NULL;

}

void mem\_free(void \* addr)

{

int i = 0;

for (auto & d : Descriptors)

{

if (d.addr <= (size\_t)addr && d.addr + settings.page\_size > (size\_t)addr && d.state != FREE)

{

auto page = Memory.begin();

advance(page, i);

// If page or several pages to be released

if (Hash\_table.find((size\_t)addr) != Hash\_table.end())

{

int pages\_to\_release = (int)(ceil)((double)(Hash\_table[(size\_t)addr]) / settings.page\_size);

auto desc = Descriptors.begin();

advance(desc, i);

while (pages\_to\_release--)

{

(\*page).clear();

page++;

(\*desc).state = FREE;

desc++;

}

Hash\_table.erase((size\_t)addr);

return;

}

// If block to be released

if ((size\_t)addr >= d.addr && (size\_t)addr <= d.addr + settings.page\_size && Hash\_table.find((size\_t)addr) != Hash\_table.end())

for (list<Block>::iterator it = (\*page).begin(); it != (\*page).end(); ++it)

{

if ((\*it).addr == (size\_t)addr)

{

int blocks\_to\_release = (int)(ceil)((double)Hash\_table[(size\_t)addr] / (\*it).size);

while (blocks\_to\_release--)

{

(\*it).state = FREE;

it++;

}

Hash\_table.erase((size\_t)addr);

// If each block is free, then release the page

for (list<Block>::iterator it2 = (\*page).begin(); it2 != (\*page).end(); ++it2)

if ((\*it2).state != FREE)

return;

d.state = FREE;

(\*page).clear();

return;

}

}

}

else if (d.addr + settings.page\_size > (size\_t)addr) // Without extra iteration

return;

i++;

}

}

void mem\_dump()

{

int i = 0;

for (auto & d : Descriptors)

{

printf("Page %d [%d..%d] \t\t \t\t %5s \n", i, d.addr, d.addr + settings.page\_size - 1, (d.state == PARTED ? "PARTED" : (d.state == FREE ? "FREE" : "BUSY")));

auto page = Memory.begin();

advance(page, get\_page\_number(d.addr));

for (auto & b : \*page)

printf("\t%5d..%5d \t\t\t %6s\n", b.addr, b.addr + b.size - 1, (b.state == FREE) ? "FREE" : "BUSY");

i++;

}

printf("\n");

}

***Файл Main.cpp***

#include "Functions.h"

#include <iostream>

using namespace std;

void menu();

int main()

{

previous\_load();

while (!cin.fail())

{

mem\_dump();

menu();

}

return 0;

}