**ЛАБОРАТОРНА РОБОТА #1**

З дисципліни «Системне програмне забезпечення»

На тему «Алокатор пам’яті загального призначення»

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

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

**Завдання:**

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.
* Після виклику mem\_free() перевіряються сусідні блоки на можливість склеювання, таким чином зменшується фрагментація пам’яті.
* 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.
* Алгоритм має окрему функцію дефрагментації, якщо фрагментація завелика.
* При дефрагментації зайняті блоки переміщуються в області молодших адрес, вільні – в області старших адрес.

**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, тому час пошуку буде залежати від кількості вже зайнятих блоків та вільних блоків, розмір яких недостатній для виділення пам’яті.

Тому час пошуку t вільного блоку пам’яті становить:

t = [tзчитування заголовку] \* [Nзайнятих блоків + Nвільних блоків із малим розміром]

Оцінка часу вивільнення зайнятого блоку:

Щоб вивільнити блок, спочатку його необхідно знайти, тобто

t = [tчас зчитування заголовку + tоперації порівняння] \* [Nзайнятих блоків + Nвільних блоків із малим розміром]

Після того відбудеться операція присвоєння статусу вільного блоку в заголовок.

Після виконання вивільнення відбудеться перевірка сусідніх блоків:

якщо склеювання неможливе, то відбудеться 4 перевірки (в коді),

якщо треба склеїти одного сусіда, то відбудеться 4 перевірки і 3 операції,

якщо треба склеїти двох сусідів, то відбудеться 4 перевірки і 6 операцій.

Тому час вивільнення залежно від вище наведених може коливатись від tmin до tmax:

tmin = [tзчитування заголовку + tоперації порівняння] \* [Nзайнятих блоків + Nвільних блоків із малим розміром] + tприсвоєння

tmax = [tзчитування заголовку + tоперації порівняння] \* [Nзайнятих блоків + Nвільних блоків із малим розміром] +3\* tприсвоєння + 2\*tдодавання + 2\*tвидалення сусіда

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

Оскільки будь-які дії із блоком реалізовуються через службові дані, тому кожен блок містить 1 байт службової інформації. Тому розмір пам’яті, необхідний для збереження службової інформації, становить:

Nслужбової інформації = [8 \* Nвсього блоків ] [біт]

**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 склеювання, що зменшує фрагментацію.
3. При зміненні розміру блоку пам’яті алокатор за допомогою багатьох перевірок пробує знайти найбільш оптимальний шлях вирішення задачі.
4. Є функція дефрагментації , коли дрібних блоків надто багато.

Недоліки:

1. При виділенні пам’яті необхідно пройти всі існуючі блоки.
2. Додаткові операції при вивільненні пам’яті щодо склеювання.
3. Багато перевірок при зміненні розміру блоку.

**5. Лістинг алокатора загального призначення.**

***Файл Functions.h***

struct Block

{

size\_t size, size\_prev, addr, occupied;

bool state;

};

struct Settings

{

size\_t memory\_size = 512; // 1024 \* 1024;

size\_t min\_size = 4;

};

enum State { FREE, BUSY };

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

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

void mem\_free(void \* addr);

void mem\_dump();

void mem\_defrag();

void previous\_load();

***Файл Functions.cpp***

#include "Functions.h"

#include <vector>

#include <iostream>

using namespace std;

vector<Block> Memory;

Settings settings;

void previous\_load()

{

Block block;

block.addr = 0;

block.size = settings.memory\_size;

block.size\_prev = 0;

block.state = FREE;

Memory.push\_back(block);

}

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

{

if (!\_size) return NULL;

size\_t size = memory\_align(\_size + 1); // + 1 byte for header info

int i = 0;

// Finding free block

for (auto b : Memory)

{

if (b.size >= size && b.state == FREE)

{

// Free block found. Creating two blocks

Block block1, block2;

block1.addr = b.addr;

block1.size = size;

block1.state = BUSY;

block1.size\_prev = b.size\_prev;

block1.occupied = \_size;

block2.addr = block1.addr + block1.size;

block2.size = b.size - block1.size;

block2.size\_prev = block1.size\_prev;

block2.state = FREE;

block2.occupied = 0;

// Add new blocks, exclude old block

Memory.insert(get\_at\_pos(i + 1), block1);

if (block2.size) Memory.insert(get\_at\_pos(i + 2), block2);

Memory.erase(get\_at\_pos(i));

return (int \*)block1.addr;

}

i++;

}

return NULL;

}

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

{

size\_t size = memory\_align(\_size);

// Find block with current address

unsigned int i = 0;

for (auto & b : Memory)

{

if (b.addr == (int)addr)

{

if (size < b.size) // Decrease block size

{

size\_t bytes\_replace = b.size - size;

if (is\_free\_space(bytes\_replace))

{

if (size == 0) {// Destroy old block

b.state = FREE;

return mem\_alloc(bytes\_replace);

}

else // Just reduce block size

{

// Create new free block and reduce old block

Block block;

block.addr = b.addr + size;

block.occupied = b.occupied - size;

block.size = b.size - size;

block.state = BUSY;

block.size\_prev = size;

b.occupied = b.size = size;

Memory.insert(get\_at\_pos(i + 1), block);

return (int \*)block.addr;

}

}

}

else if (size > b.size)

{

// Try to extend block to neighbour

size\_t free\_memory = 0, bytes\_to\_extend = size - b.size;

free\_memory += (i && (\*get\_at\_pos(i - 1)).state == FREE) ? (\*get\_at\_pos(i - 1)).size : 0;

free\_memory += (i + 1 < Memory.size() && (\*get\_at\_pos(i + 1)).state == FREE ? (\*get\_at\_pos(i + 1)).size : 0);

if (free\_memory >= bytes\_to\_extend)

{

// use neighbour's memory

if (i && (\*get\_at\_pos(i - 1)).state == FREE)

if ((\*get\_at\_pos(i - 1)).size <= bytes\_to\_extend)

{

b.size += (\*get\_at\_pos(i - 1)).size;

b.addr = (\*get\_at\_pos(i - 1)).addr;

bytes\_to\_extend -= (\*get\_at\_pos(i - 1)).size;

(\*get\_at\_pos(i - 1)).size = 0;

}

else

{

(\*get\_at\_pos(i - 1)).size -= bytes\_to\_extend;

b.size += bytes\_to\_extend;

b.addr -= bytes\_to\_extend;

bytes\_to\_extend = 0;

}

if (bytes\_to\_extend)

if ((\*get\_at\_pos(i + 1)).size <= bytes\_to\_extend)

{

b.size += (\*get\_at\_pos(i + 1)).size;

(\*get\_at\_pos(i + 1)).size -= bytes\_to\_extend;

}

else

{

(\*get\_at\_pos(i + 1)).size -= bytes\_to\_extend;

(\*get\_at\_pos(i + 1)).addr += bytes\_to\_extend;

b.size += bytes\_to\_extend;

}

if (!(\*get\_at\_pos(i - 1)).size) Memory.erase(get\_at\_pos(i - 1));

if (!(\*get\_at\_pos(i)).size) Memory.erase(get\_at\_pos(i));

return (int \*)(\*get\_at\_pos(i - 1)).addr;

}

else

{

// Neighbours can't help, create new block in memory

b.state = FREE;

b.occupied = 0;

return mem\_alloc(size);

}

}

}

i++;

}

return NULL;

}

void mem\_free(void \* addr)

{

unsigned int i = 0;

for (auto & b : Memory)

{

if (b.addr == (int)addr)

{

b.state = FREE;

// If there is free neighbour besides

if (i && (\*get\_at\_pos(i - 1)).state == FREE)

{

(\*get\_at\_pos(i - 1)).size += b.size;

Memory.erase(get\_at\_pos(i));

}

if (i + 1 < Memory.size() && (\*get\_at\_pos(i + 1)).state == FREE)

{

b.size += (\*get\_at\_pos(i + 1)).size;

Memory.erase(get\_at\_pos(i + 1));

}

return;

}

i++;

}

}

void defragment(int i)

{

vector<Block>::iterator & it1 = get\_at\_pos(i), it2 = get\_at\_pos(i + 1);

if (it1 == Memory.end() || it2 == Memory.end()) return;

if ((\*it1).state == FREE && (\*it2).state == FREE)

{

// Glue both free blocks

(\*it1).size = (\*it1).size + (\*it2).size;

Memory.erase(it2);

return defragment(i);

}

else if ((\*it1).state == FREE && (\*it2).state == BUSY)

{

// Swap free and busy blocks

Block block;

block.addr = (\*it1).addr + (\*it2).size;

block.occupied = (\*it1).addr;

block.size = (\*it1).size;

block.size\_prev = (\*it1).size\_prev;

block.state = (\*it1).state;

(\*it2).addr -= block.size;

Memory.erase(it1);

Memory.insert(get\_at\_pos(i + 1), block);

return defragment(i);

}

return defragment(i + 1);

}

void mem\_defrag()

{

defragment(0); // Recursive

}

void mem\_dump()

{

cout << endl << "Memory map:" << endl;

for (auto b : Memory)

printf("%6d%4s%6d\t%30s\n", b.addr, "..", b.size-1 + b.addr, (b.state == FREE ? "FREE" : "BUSY"));

}

***Файл Main.cpp***

#include "Functions.h"

#include <iostream>

using namespace std;

void menu();

int main()

{

previous\_load();

while (true)

{

mem\_dump();

menu();

}

return 0;

}

**6. Приклад роботи алокатора.**

