МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

Сучасні операційні системи

Лабораторна робота №1. Аллокатор пам’яті загального призначення

*Виконав студент гр. ІС-72*

*Кривохижа Р. А.*

Київ

НТУУ «КПІ»

2020

**Мета роботи:** розробити аллокатор загального призначення.

**Завдання:**

Розробити аллокатор загального призначення, беручи до уваги наступні умови:

* Області пам'яті можна виділяти будь-яким доступним способом.
* Функції mem\_alloc (), mem\_realloc () і mem\_free () повинні відповідати наведеним прототипам.
* Адреси пам'яті, які повертаються функціями mem\_alloc() і mem\_realloc(), повинні бути вирівняні на границю 4 байт.
* Оптимізувати час пошуку вільного блоку пам'яті і час звільнення зайнятого блоку.
* Написати функцію, яка повинна виводити на консоль стан областей пам'яті.

**Опис розробленого алгоритму:**

* Спочатку иділення пам’яті через створення масиву:

int \*mas = new int[n+1];

Масив типу int, щоб спростити вирівнювання блоків на границю в 4 байти.

Кожен блок виділеної пам’яті містить заголовок BlockHeader. Заголовок займає 12 байт. Під час ініціалізації виділяється 1 вільний блок, який займає всю доступну пам’ять масиву.

* Виділення пам’яті заданого розміру mem\_alloc(size\_t size):

Вибирається перший знайдений підходящий блок, тобто такий, розмір якого не менший за size. Якщо розмір більший необхідного, блок розбивається на зайнятий і вільний блоки, і користувачу повертається вказівник на початок зайнятого блоку. Якщо потрібний блок не знайдений, то повертається NULL.

* Перевиділення пам’яті mem\_realloc(void \*addr, size\_t size).

Якщо addr = NULL, то виконується виклик mem\_alloc(size). Інакше, відбувається перевірка сусідніх блоків. Якщо хоча б один з них вільний, відбувається об’єднання блоків. Далі перевіряється, чи цей новий блок достатнього розміру для перевизначення. Якщо так, то в ньому створюється 2 блоки: зайнятий і вільний. У зайнятий копіюються дані з колишнього блоку користувача. Якщо ж ні – йде пошук нового вільного блоку і, якщо він знайдений, виділяється блок для користувача і копіюються данні. Інакше – повертається NULL, а данні користувача не змінюютсья.

* Звільнення пам’яті mem\_free(void \*addr). Помічаємо блок як вільний і об’єднуємо даний блок с сусідніми вільними блоками (максимум – 2).

**Розроблене програмне забезпечення:**

// Test.cpp

#include <iostream>

#include "Allocator.h"

using namespace std;

void fillBlock(void \*start, int size, int filler)

{

  for (int i = 0; i < size; i++)

  {

    \*((int \*)start + 1) = filler;

  }

}

void test()

{

  cout << "test() started! \n";

  const int n = 2000;

  const int calls = 15;

  const int bSize = 50;

  Allocator al = Allocator(n);

  void \*curBlock;

  void \*mas[calls]; // has all user blocks

  for (int i = 0; i < calls; i++)

  {

    curBlock = al.mem\_alloc(bSize);

    mas[i] = curBlock;

    fillBlock(curBlock, 50, 170); //170(dec) = 1010 1010(bin)

  }

  cout << "alloc 15 blocks (length = 50) \n";

  al.mem\_dump();

  cout << "min 3th el to 20\n";

  al.mem\_realloc(mas[3], 20);

  al.mem\_dump();

  cout << "max 3th el to 30\n";

  al.mem\_realloc(mas[3], 30);

  al.mem\_dump();

  cout << "max 3th el to 49\n";

  al.mem\_realloc(mas[3], 49);

  al.mem\_dump();

  cout << "free 2th and 4th els \n";

  al.mem\_free(mas[2]);

  al.mem\_free(mas[4]);

  al.mem\_dump();

  cout << "...and max 3th el to 156 \n";

  al.mem\_realloc(mas[3], 156);

  al.mem\_dump();

  cout << "min 2th el to 53 \n";

  al.mem\_realloc(mas[2], 53);

  al.mem\_dump();

  cout << "test() finished! \n";

}

int main()

{

  test();

  getchar();

  return 0;

}

// Allocator.cpp

#include <Windows.h>

#include <iostream>

#include "Allocator.h"

using namespace std;

Allocator::Allocator(const int n)

{

  int \*mas = new int[n + 1];

  N = n;

  bSize = sizeof(BlockHeader) / sizeof(int);

  begin = (BlockHeader \*)(&mas[0]);

  begin->prevsize = NULL;

  begin->size = &mas[n] - &mas[0] - bSize;

  begin->state = false;

  endOfMemory = &mas[n];

}

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

{

  BlockHeader \*current = begin;

  while (current->size < size || current->state)

  {

    if (isLast(current))

      return NULL;

    current = nextBlockHeader(current);

  }

  if ((current->size == size) || ((current->size - size) < bSize))

  {

    current->state = 1;

    return getBlock(current);

  }

  else

  {

    return separateOnUseAndFree(current, size);

  }

}

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

{

  if (addr == NULL)

  {

    return mem\_alloc(size);

  }

  BlockHeader \*current = (BlockHeader \*)addr - 1;

  size\_t deltaSize = current->size - size; // size\_t - unsigned int!!! fix it!

  if (deltaSize == 0)

    return addr;

  BlockHeader \*previous = previousBlockHeader(current);

  BlockHeader \*next = nextBlockHeader(current);

  if (previous == NULL)

    if (next == NULL)

    {

      //prev=NULL, next=NULL

      return searchNewBlock(addr, size);

    }

    else if (next->state)

      //prev=NULL, next=1

      return searchNewBlock(addr, size);

    else

      //prev=NULL, next=0

      return expandRight(addr, size); //check later - fixed

  else if (previous->state)

    if (next == NULL)

      //prev=1, next=NULL

      return searchNewBlock(addr, size);

    else if (next->state)

      //prev=1, next=1

      return searchNewBlock(addr, size);

    else

      //prev=1, next=0

      return expandRight(addr, size);

  else if (next == NULL)

    //prev=0, next=NULL

    return expandLeft(addr, size);

  else if (next->state)

    //prev=0, next=1

    return expandLeft(addr, size);

  else

    //prev=0, next=0

    return expandBoth(addr, size);

  return NULL;

}

void Allocator::mem\_free(void \*addr)

{

  BlockHeader \*current = (BlockHeader \*)addr - 1;

  BlockHeader \*next = nextBlockHeader(current);

  BlockHeader \*previous = previousBlockHeader(current);

  if (previous == NULL)

    if (next == NULL)

      //prev=NULL, next=NULL

      current->state = false;

    else if (next->state)

      //prev=NULL, next=1

      current->state = false;

    else

      //prev=NULL, next=0

      mergeWithNext(current, next);

  else if (previous->state)

    if (next == NULL)

      //prev=1, next=NULL

      current->state = false;

    else if (next->state)

      //prev=1, next=1

      current->state = false;

    else

      //prev=1, next=0

      mergeWithNext(current, next);

  else if (next == NULL)

    //prev=0, next=NULL

    mergeWithPrevious(previous, current);

  else if (next->state)

    //prev=0, next=1

    mergeWithPrevious(previous, current, next);

  else

    //prev=0, next=0

    mergeBoth(previous, current, next);

}

BlockHeader \*Allocator::nextBlockHeader(BlockHeader \*current)

{

  if (isLast(current))

    return NULL;

  return (BlockHeader \*)((int \*)((char \*)current + sizeof(BlockHeader)) + current->size);

}

BlockHeader \*Allocator::previousBlockHeader(BlockHeader \*current)

{

  if (current->prevsize == NULL)

    return NULL;

  return (BlockHeader \*)((int \*)((char \*)current - sizeof(BlockHeader)) - current->prevsize);

}

bool Allocator::isLast(BlockHeader \*h)

{

  if (((int \*)((char \*)h + sizeof(BlockHeader)) + h->size) == endOfMemory)

    return true;

  else

    return false;

}

void Allocator::mergeWithNext(BlockHeader \*current, BlockHeader \*next)

{

  current->size += next->size + sizeof(BlockHeader) / sizeof(int);

  current->state = false;

  BlockHeader \*next2 = nextBlockHeader(next);

  if (next2 != NULL)

    next2->prevsize = current->size;

}

void Allocator::mergeWithPrevious(BlockHeader \*previous, BlockHeader \*current)

{

  previous->size += current->size + sizeof(BlockHeader) / sizeof(int);

}

void Allocator::mergeWithPrevious(BlockHeader \*previous, BlockHeader \*current, BlockHeader \*next)

{

  previous->size += current->size + sizeof(BlockHeader) / sizeof(int);

  next->prevsize = previous->size;

}

void Allocator::mergeBoth(BlockHeader \*previous, BlockHeader \*current, BlockHeader \*next)

{

  previous->size += current->size + next->size + 2 \* sizeof(BlockHeader) / sizeof(int);

  BlockHeader \*next2 = nextBlockHeader(next);

  if (next2 != NULL)

    next2->prevsize = previous->size;

}

void Allocator::copyData(void \*from, void \*to, size\_t quantity)

{

  int \*f = (int \*)from;

  int \*t = (int \*)to;

  for (unsigned int i = 0; i < quantity; i++)

    t[i] = f[i];

}

void \*Allocator::searchNewBlock(void \*addr, size\_t size)

{

  BlockHeader \*current = (BlockHeader \*)addr - 1;

  size\_t deltaSize = current->size - size;

  if ((deltaSize > 0) && (deltaSize < 3))

  {

    return addr;

  }

  if (deltaSize > 0)

  {

    return separateOnUseAndFree((BlockHeader \*)addr - 1, size);

  }

  else

  {

    void \*nBlock = mem\_alloc(size);

    if (nBlock != NULL)

    {

      copyData(addr, nBlock, current->size);

      mem\_free(addr);

      return nBlock;

    }

    return NULL;

  }

}

void \*Allocator::expandLeft(void \*addr, size\_t size)

{

  BlockHeader \*current = (BlockHeader \*)addr - 1;

  BlockHeader \*previous = previousBlockHeader(current);

  size\_t area = current->size + current->prevsize + sizeof(BlockHeader) / sizeof(int);

  size\_t deltaSize = current->size - size;

  if (deltaSize > 0)

  { //minimize

    //copy

    int \*first = (int \*)addr;

    for (int i = size - 1; i >= 0; i--)

    {

      first[i + deltaSize] = first[i];

    }

    //set BlockHeaders

    previous->size += deltaSize;

    current = nextBlockHeader(previous);

    initBlockHeader(current, true, previous->size, size, 7);

    BlockHeader \*next = nextBlockHeader(current);

    if (next != NULL)

      next->prevsize = current->size;

    return getBlock(current);

  }

  else

  { //maximize

    if (area >= size)

      if ((area == size) || ((area - size) < 3))

      {

        //set BH

        previous->size = area;

        BlockHeader \*next = nextBlockHeader(current);

        if (next != NULL)

          next->prevsize = previous->size;

        previous->state = true;

        //copy

        copyData(addr, getBlock(previous), current->size);

        return getBlock(previous);

      }

      else

      {

        previous->size = size;

        previous->state = true;

        copyData(addr, getBlock(previous), current->size);

        current = nextBlockHeader(previous);

        current->size = area - size - sizeof(BlockHeader) / sizeof(int);

        current->state = false;

        current->prevsize = previous->size;

        BlockHeader \*next = nextBlockHeader(current);

        if (next != NULL)

          next->prevsize = current->size;

        return getBlock(previous);

      }

    else

    {

      void \*p = mem\_alloc(size);

      if (p != NULL)

      {

        copyData(addr, p, current->size);

        BlockHeader \*next = nextBlockHeader(current);

        if (next == NULL)

          mergeWithPrevious(previous, current);

        else

          mergeWithPrevious(previous, current, next);

        return p;

      }

    }

  }

  return NULL;

}

void \*Allocator::expandRight(void \*addr, size\_t size)

{

  BlockHeader \*current = (BlockHeader \*)addr - 1;

  BlockHeader \*next = nextBlockHeader(current);

  size\_t area = current->size + next->size + sizeof(BlockHeader) / sizeof(int);

  int deltaSize = (int)(current->size) - (int)(size); //TODO: fis expression to int - fixed

  if (deltaSize > 0)

  { //minimize

    //set BlockHeaders

    current->size = size;

    next = nextBlockHeader(current);

    initBlockHeader(next, false, current->size, area - size - sizeof(BlockHeader) / sizeof(int), 7);

    BlockHeader \*next2 = nextBlockHeader(next);

    if (next2 != NULL)

      next2->prevsize = next->size;

    return getBlock(current);

  }

  else

  { //maximize

    if (area >= size)

      if ((area - size) < 3)

      {

        //set BH

        current->size = area;

        next = nextBlockHeader(current);

        if (next != NULL)

          next->prevsize = current->size;

        return getBlock(current);

      }

      else

      {

        current->size = size;

        next = nextBlockHeader(current);

        next->size = area - size - sizeof(BlockHeader) / sizeof(int);

        next->prevsize = size;

        next->state = false;

        BlockHeader \*next2 = nextBlockHeader(next);

        if (next2 != NULL)

          next2->prevsize = next->size;

        return getBlock(current);

      }

    else

    {

      void \*p = mem\_alloc(size);

      if (p != NULL)

      {

        copyData(addr, p, current->size);

        next = nextBlockHeader(current);

        mergeWithNext(current, next);

        return p;

      }

    }

  }

  return NULL;

}

void \*Allocator::expandBoth(void \*addr, size\_t size)

{

  BlockHeader \*current = (BlockHeader \*)addr - 1;

  BlockHeader \*previous = previousBlockHeader(current);

  BlockHeader \*next = nextBlockHeader(current);

  size\_t area = current->size + previous->size + next->size + 2 \* sizeof(BlockHeader) / sizeof(int);

  int deltaSize = (int)(current->size) - (int)(size); //fix unsigned - fixed

  if (deltaSize > 0)

  { //minimize

    mergeBoth(previous, current, next);

    //copy

    copyData(addr, getBlock(previous), size);

    return separateOnUseAndFree(previous, size);

  }

  else

  { //maximize

    if (area >= size)

      if ((area == size) || ((area - size) < 3))

      {

        //set BH

        previous->size = area;

        BlockHeader \*next2 = nextBlockHeader(next);

        if (next2 != NULL)

          next2->prevsize = previous->size;

        previous->state = true;

        //copy

        copyData(addr, getBlock(previous), current->size);

        return getBlock(previous);

      }

      else

      {

        mergeBoth(previous, current, next);

        //copy

        copyData(addr, getBlock(previous), current->size);

        return separateOnUseAndFree(previous, size);

      }

    else

    {

      void \*p = mem\_alloc(size);

      if (p != NULL)

      {

        copyData(addr, p, current->size);

        mergeBoth(previous, current, next);

        return p;

      }

    }

  }

  return NULL;

}

void Allocator::initBlockHeader(BlockHeader \*bh, bool state, size\_t previous, size\_t size, int mask)

{

  if (bh == NULL)

    return;

  if (mask & 4)

  {

    bh->state = state;

  }

  if (mask & 2)

  {

    bh->prevsize = previous;

  }

  if (mask & 1)

    bh->size = size;

}

void \*Allocator::separateOnUseAndFree(BlockHeader \*current, size\_t size)

{

  size\_t curSize = current->size;

  BlockHeader \*next = nextBlockHeader(current);

  if (next == NULL)

  {

    initBlockHeader(current, true, NULL, size, 5);

    BlockHeader \*next1 = nextBlockHeader(current);

    initBlockHeader(next1, false, size, (curSize - size - bSize), 7);

    return getBlock(current);

  }

  else

  {

    initBlockHeader(current, true, NULL, size, 5);

    BlockHeader \*next1 = nextBlockHeader(current);

    initBlockHeader(next1, false, size, (curSize - size - bSize), 7);

    next->prevsize = next1->size;

    return getBlock(current);

  }

}

void \*Allocator::getBlock(BlockHeader \*h)

{

  return (void \*)(h + 1);

}

bool Allocator::checkDamage(int filler)

{

  BlockHeader \*current = begin;

  int count = 0;

  while (true)

  {

    if (isLast(current))

      break;

    void \*start = getBlock(current);

    for (unsigned int i = 0; i < current->size; i++)

    {

      if (\*((int \*)start + i) != filler)

      {

        count++;

      }

    }

    current = nextBlockHeader(current);

  }

  if (count)

  {

    cout << "damaged: " << endl;

    return true;

  }

  else

  {

    return false;

  }

}

void Allocator::mem\_dump()

{

  BlockHeader \*current = begin;

  int i = 0;

  cout << "--- Out all BlockHeaders:" << endl;

  while (current != NULL)

  {

    cout << i << ". " << current << "   " << current->state << "   " << current->size << "   " << current->prevsize << endl;

    i++;

    current = nextBlockHeader(current);

  }

  cout << "--- --- --- - --- --- ---" << endl;

}

// Allocator.h

#pragma once

struct BlockHeader

{

  bool state; //1 - used, 0 - unused

  size\_t size;

  size\_t prevsize;

};

class Allocator

{

public:

  Allocator(const int n);

  //return addr on begin of allocated block or NULL

  void \*mem\_alloc(size\_t size);

  //return addr on begin of reallocated block or NULL

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

  //free block by this address

  void mem\_free(void \*addr);

  //out blocks characteristic in table on console

  void mem\_dump();

private:

  size\_t bSize;       //struct BlockHeader size in int

  int N;              // length all memory in int

  BlockHeader \*begin; //first block

  int \*endOfMemory;   //last int in memory

  //all blocks must be fill the same number

  bool checkDamage(int filler);

  //return next BH or NULL if it block is last

  BlockHeader \*nextBlockHeader(BlockHeader \*current);

  //return previous BH or NULL if it block is first

  BlockHeader \*previousBlockHeader(BlockHeader \*current);

  //check if endOfMemory belongs to this block

  bool isLast(BlockHeader \*h);

  //next 4 functions merge 2 or 3 free blocks

  void mergeWithNext(BlockHeader \*current, BlockHeader \*next);

  void mergeWithPrevious(BlockHeader \*previous, BlockHeader \*current);

  void mergeWithPrevious(BlockHeader \*previous, BlockHeader \*current, BlockHeader \*next);

  void mergeBoth(BlockHeader \*previous, BlockHeader \*current, BlockHeader \*next);

  //copy data in new block (all  or part = length of new block)

  void copyData(void \*from, void \*to, size\_t quantity);

  //return link on finded free block or NULL

  void \*searchNewBlock(void \*addr, size\_t size);

  //next founctions merge 2 or 3 blocks (one is use), then separate them on use and free and copy data to new use

  void \*expandLeft(void \*addr, size\_t size);

  void \*expandRight(void \*addr, size\_t size);

  void \*expandBoth(void \*addr, size\_t size);

  //set fields of BH selected by mask

  void initBlockHeader(BlockHeader \*bh, bool state, size\_t previous, size\_t size, int mask); //mask 7 (binary: 111) - all

  //size - length of new use block

  void \*separateOnUseAndFree(BlockHeader \*current, size\_t size);

  //return begin of memory block for user (after BH)

  void \*getBlock(BlockHeader \*h);

};

**Результати роботи:**





