Міністерство освіти і науки України

Національний технічний університет України

«Київський політехнічний інститут»

Факультет прикладної математики

**Лабораторна робота №2**

**з дисципліни: «**Системне Програмне Забезпечення**»**

|  |  |
| --- | --- |
| Виконав:  Студент групи КВ-92  Степанюк Михайло |  |

Київ 2012

**Постановка задачі:**

**Алокатор пам’яті загального призначення (частина 2)**

Розробити алокатор загального призначення, використовуючи за основу описаний вище базовий варіант алгоритму, приймаючи до уваги наступні умови:

1. Області пам’яті можна виділяти будь-яким доступним способом.
2. Функції mem\_alloc(), mem\_realloc() та mem\_free() повинні відповідати приведеним вище прототипам.
3. Адреси пам’яті, що повертаються функціями mem\_alloc() та mem\_realloc(), повинні бути вирівняні на межу в 4 байти.
4. Спробувати зменшити час пошуку вільного блоку пам’яті та час звільнення занятого блоку.
5. Спробувати зменшити фрагментацію пам’яті.
6. Написати функцію mem\_dump(), яка повинна виводити на консоль стан областей пам’яті.

**Код програми:**

MemMgr.h

#ifndef \_MEMMGR\_H

#define \_MEMMGR\_H

#include<iostream>

#include<queue>

#include<vector>

#include<map>

#pragmawarning(disable : 4018)

usingnamespacestd;

typedefunsignedcharuchar, \*PUCHAR;

staticucharsize\_of\_st = sizeof(size\_t);

enum { defsize = 64\*1024};

classMemMgr

{

private:

vector<uchar\*>m\_begin; // указатель на начало памяти

MemMgr(constMemMgr&);

MemMgr&operator = (constMemMgr&);

public:

MemMgr();

~MemMgr();

void\* alloc(size\_t bytes);

void\* realloc( void\*, size\_t);

voidfree(void\*);

voidmem\_dump();

};

#endif

**MemMgr.cpp**

#include "MemMgr.h"

MemMgr::MemMgr()

{

uchar\* new\_block = newuchar[defsize + 3\*size\_of\_st];

m\_begin.push\_back( new\_block );

size\_t\* tmp = (size\_t \*) \*(m\_begin.end()-1);

\*tmp = defsize; //curr\_len

tmp += size\_of\_st;

\*tmp = 0; //prev\_bloc\_len

tmp += size\_of\_st;

\*tmp = 0; // 0 - free;

}

void\* MemMgr::alloc(size\_t bytes)

{

void \*memory = 0;

if(bytes == 0)

returnmemory;

bytes = (bytes + 3) & 0xFFFFFFFC;

if(bytes>defsize)

return NULL;

vector<uchar\*>::iteratorit;

for(it = m\_begin.begin(); it<m\_begin.end(); it++)

{

size\_t\* tmp = (size\_t \*) \*it;

size\_t nch\_size = 0;

while(nch\_size<defsize)

{

size\_t block\_len = \*tmp;

tmp += 2\*size\_of\_st;

if(\*tmp == 0 &&block\_len >= bytes)

{

\*tmp = 1; //settofilled

tmp += size\_of\_st;

memory = (void \*)tmp;

tmp-= 3\*size\_of\_st; //setnumberofbytesthatfilled

\*tmp = bytes;

//createfreeblock

if(block\_len - bytes> 3\*size\_of\_st)

{

tmp += 3\*size\_of\_st + bytes/4;

\*tmp = block\_len - bytes - 3\*size\_of\_st;

tmp += size\_of\_st;

\*tmp = bytes;

tmp += size\_of\_st;

\*tmp = 0;

tmp += size\_of\_st;

}

break;

}

tmp += size\_of\_st + block\_len/4;

nch\_size += 3\*sizeof(size\_t) + block\_len;

}

}

if(memory == 0)

{

try

{

m\_begin.push\_back( newuchar[defsize + 3\*size\_of\_st] );

size\_t\* tmp = (size\_t \*) \*(m\_begin.end() - 1);

\*tmp = bytes;

tmp += size\_of\_st;

\*tmp = 0;

tmp += size\_of\_st;

\*tmp = 1; // 0 - free;

tmp += size\_of\_st;

memory = tmp;

if(defsize - bytes> 3\*size\_of\_st)

{

tmp += size\_of\_st + bytes/4;

//freespace

\*tmp = defsize - bytes - 3\*size\_of\_st;

tmp += size\_of\_st;

\*tmp = bytes;

tmp += size\_of\_st;

\*tmp = 0;

}

}

catch(...)

{ memory = 0; }

}

returnmemory;

}

void\* MemMgr::realloc(void\* spase, size\_t new\_size)

{

if(new\_size == 0)

return NULL;

new\_size = (new\_size + 3) & 0xFFFFFFFC;

void\* memory = 0;

if(spase == 0)

returnalloc(new\_size);

size\_t\* tmp = (size\_t \*) spase;

tmp -= 3\*size\_of\_st;

size\_t curr\_size = \*tmp;

if(curr\_size == new\_size)

returnspase;

elseif(curr\_size >new\_size )

{

memory = spase;

if(curr\_size - new\_size>= 3\*size\_of\_st )

{

tmp = (size\_t \*)spase - 2\*size\_of\_st;

size\_t prev\_size = \*tmp;

size\_t left\_b = prev\_size;

size\_t total\_len = curr\_size + 3\*size\_of\_st;

while(left\_b != 0)

{

total\_len += left\_b + 3\*size\_of\_st;

tmp -= (left\_b/4 + 3\*size\_of\_st);

left\_b = \*tmp;

}

if((defsize + 3\*size\_of\_st) - total\_len > 3\*size\_of\_st) {

//checkrightblock

tmp = (size\_t \*)spase + curr\_size/4;

size\_t\* alien\_block = tmp;

tmp += 2\*size\_of\_st;

if(\*tmp == 0) // freememory + right\_block

{

tmp -= 2\*size\_of\_st;

size\_t right\_len = \*tmp;

if((defsize + 3\*size\_of\_st) - total\_len - right\_len > 6\*size\_of\_st )

alien\_block = tmp + right\_len/4 + 3\*size\_of\_st;

else

alien\_block = NULL;

tmp = (size\_t \*)spase - 3\*size\_of\_st;

\*tmp = new\_size;

tmp += new\_size/4 + 3\*size\_of\_st;

\*tmp = curr\_size - new\_size + right\_len;

tmp += size\_of\_st;

\*tmp = new\_size;

tmp += size\_of\_st;

\*tmp = 0;

if(alien\_block)

{

alien\_block += size\_of\_st;

\*alien\_block = curr\_size - new\_size + right\_len;

}

}

else

{

tmp = (size\_t \*)spase;

tmp -= 3\*size\_of\_st;

\*tmp = new\_size;

alien\_block += size\_of\_st;

\*alien\_block = new\_size;

}

}

else

{

tmp = (size\_t \*)spase;

tmp -= 3\*size\_of\_st;

\*tmp = new\_size;

tmp += new\_size/4 + 3\*size\_of\_st;

\*tmp = curr\_size - new\_size - 3\*size\_of\_st;

tmp += size\_of\_st;

\*tmp = new\_size;

tmp += size\_of\_st;

\*tmp = 0;

tmp += size\_of\_st;

}

}

returnmemory;

}

else

{

//checknextblock (is\_free ??)

tmp += size\_of\_st;

size\_t prev\_size = \*tmp;

size\_t left\_b = prev\_size;

size\_t total\_len = curr\_size + 3\*size\_of\_st;

while(left\_b != 0)

{

total\_len += left\_b + 3\*size\_of\_st;

tmp -= (left\_b/4 + 3\*size\_of\_st);

left\_b = \*tmp;

}

if((defsize + 3\*size\_of\_st) - total\_len > 3\*size\_of\_st) //rightblockisreal !

{

//checkrightblock

tmp = (size\_t \*)spase + curr\_size/4;

size\_t right\_len = \*tmp;

tmp += 2\*size\_of\_st;

if(\*tmp == 0 &&new\_size<= (curr\_size + right\_len + 3\*size\_of\_st))

{

memory = spase;

tmp = (size\_t \*)spase - 3\*size\_of\_st;

\*tmp = new\_size;

//newfreeblock

if(curr\_size + right\_len >new\_size)

{

tmp += new\_size/4 + 3\*size\_of\_st;

\*tmp = curr\_size + right\_len - new\_size;

tmp += size\_of\_st;

\*tmp = new\_size;

tmp += size\_of\_st;

\*tmp = 0;

tmp += size\_of\_st;

}

}

}

if(memory == 0)

{

//findnewblock

memory = alloc(new\_size);

if(memory != NULL)

{

uchar\* old\_mem = (uchar \*)spase;

uchar\* new\_mem = (uchar \*)spase;

//copydatafromoldblocktonew;

for(int i = 0; i <curr\_size; i++ )

{

\*new\_mem = \*old\_mem;

new\_mem += sizeof(uchar);

old\_mem += sizeof(uchar);

}

//clearoldmemory

free(spase);

}

}

returnmemory;

}

}

voidMemMgr::free(void\* space)

{

size\_t\* alien\_block = NULL;

size\_t\* tmp = (size\_t \*)space - size\_of\_st;

\*tmp = 0; //settofree

tmp -= size\_of\_st;

size\_t prev\_size = \*tmp;

tmp -= size\_of\_st;

size\_t curr\_size = \*tmp;

tmp += size\_of\_st;

////////////////

size\_t left\_b = prev\_size;

size\_t total\_len = curr\_size + 3\*size\_of\_st;

while(left\_b != 0)

{

total\_len += left\_b + 3\*size\_of\_st;

tmp -= (left\_b/4 + 3\*size\_of\_st);

left\_b = \*tmp;

}

if((defsize + 3\*size\_of\_st) - total\_len > 3\*size\_of\_st) // rightblockisreal !

{

//checkrightblock

tmp = (size\_t \*)space + curr\_size/4;

alien\_block = tmp;

tmp += 2\*size\_of\_st;

if(\*tmp == 0)

{

//right + current

tmp -= 2\*size\_of\_st;

size\_t right\_len = \*tmp;

if((defsize + 3\*size\_of\_st) - total\_len - right\_len > 6\*size\_of\_st )

alien\_block = tmp + right\_len/4 + 3\*size\_of\_st;

else

alien\_block = NULL;

tmp = (size\_t \*)space - 3\*size\_of\_st;

\*tmp += right\_len + 3\*size\_of\_st;

curr\_size += right\_len + 3\*size\_of\_st;

}

}

////////////////

if(prev\_size != 0)

{

tmp = (size\_t \*)space - prev\_size/4 - 4\*size\_of\_st;

if(\*tmp == 0) //prevblockisfree!

{

tmp -= 2\*size\_of\_st;

\*tmp += curr\_size + 3\*size\_of\_st;

curr\_size = \*tmp;

}

}

if(alien\_block)

{

alien\_block += size\_of\_st;

\*alien\_block = curr\_size;

}

}

MemMgr::~MemMgr(void)

{

for(size\_t i = 0; i <m\_begin.size(); i++)

{

delete[] \*(m\_begin.end()-1);

m\_begin.pop\_back();

} }

voidMemMgr::mem\_dump()

{

vector<uchar\*>::iteratorit;

int i = 1;

cout<<endl<<endl;

for(it = m\_begin.begin(); it<m\_begin.end(); it++)

{

cout<<"Page "<<i<<" Attr:"<<endl;

i++;

size\_t\* tmp = (size\_t \*) \*it;

size\_t len = 0;

while(len<defsize)

{

size\_t block\_len = \*tmp;

tmp += size\_of\_st;

size\_t prev\_len = \*tmp;

tmp += size\_of\_st;

size\_t is\_free = \*tmp;

len += block\_len + 3\*size\_of\_st;

tmp += block\_len/4 + size\_of\_st;

cout<<"Blocklen = "<<block\_len<<"; prev = "<<prev\_len;

if(is\_free == 0)

cout<<"; is\_free = true"<<endl;

else

cout<<"; is\_free = false"<<endl;

}

}

}

Test.cpp

#include "MemMgr.h"

//MemMgrtest

staticMemMgrmyMemMgr;

intmain()

{

size\_t i = 7;

//cout<<defsize<<endl;

cout<<"alloc j (28 bytes)"<<endl;

int\* j = (int \*) myMemMgr.alloc( i\*sizeof(int) );

myMemMgr.mem\_dump();

cout<<"\n"<<endl;

cout<<"realloc j (84 bytes)"<<endl;

j = (int \*) myMemMgr.realloc( j, (i+i+i)\*sizeof(int) );

myMemMgr.mem\_dump();

cout<<"\n"<<endl;

cout<<"alloc l (84 bytes)"<<endl;

int\* l = (int \*) myMemMgr.alloc((i+i+i)\*sizeof(int) );

myMemMgr.mem\_dump();

cout<<"\n"<<endl;

cout<<"free l (84 bytes)"<<endl;

myMemMgr.free(l);

myMemMgr.mem\_dump();

cout<<"\n"<<endl;

cout<<"realloc j (84 bytes)"<<endl;

j = (int \*) myMemMgr.realloc( j, (i)\*sizeof(int) );

myMemMgr.mem\_dump();

cout<<"\n"<<endl;

cout<<"allocdefsize (65536 bytes)"<<endl;

uchar\* p = (uchar \*) myMemMgr.alloc(defsize);

myMemMgr.mem\_dump();

cout<<"\n"<<endl;

cout<<"realloc p = defsize - 500 (64536 bytes)"<<endl;

p = (uchar \*) myMemMgr.realloc( p, defsize - 1000);

myMemMgr.mem\_dump();

cout<<"\n"<<endl;

cout<<"realloc p (65536 bytes)"<<endl;

p = (uchar \*) myMemMgr.realloc( p, defsize );

myMemMgr.mem\_dump();

cout<<"\n"<<endl;

cout<<"free p (65536 bytes)"<<endl;

myMemMgr.free( (void \*)p );

myMemMgr.mem\_dump();

cout<<"\n"<<endl;

cout<<"free j (84 bytes)"<<endl;

myMemMgr.free( (void \*)j );

myMemMgr.mem\_dump();

cin.get();

return 0;

}

**Результат роботи:**

**alloc j (28 bytes)**

**Page #1 attributes:**

**len = 28; prev = 0;free = FALSE**

**len = 65496; prev = 28;free = TRUE**

**realloc j (84 bytes)**

**Page #1 attributes:**

**len = 84; prev = 0;free = FALSE**

**len = 65440; prev = 84;free = TRUE**

**alloc l (84 bytes)**

**Page #1 attributes:**

**len = 84; prev = 0;free = FALSE**

**len = 84; prev = 84;free = FALSE**

**len = 65344; prev = 84;free = TRUE**

**free l (84 bytes)**

**Page #1 attributes:**

**len = 84; prev = 0;free = FALSE**

**len = 65440; prev = 84;free = TRUE**

**realloc j (84 bytes)**

**Page #1 attributes:**

**len = 28; prev = 0;free = FALSE**

**len = 65496; prev = 28;free = TRUE**

**allocdefsize (65536 bytes)**

**Page #1 attributes:**

**len = 28; prev = 0;free = FALSE**

**len = 65496; prev = 28;free = TRUE**

**Page #2 attributes:**

**len = 65536; prev = 0;free = FALSE**

**realloc p = defsize - 500 (64536 bytes)**

**Page #1 attributes:**

**len = 28; prev = 0;free = FALSE**

**len = 65496; prev = 28;free = TRUE**

**Page #2 attributes:**

**len = 65036; prev = 0;free = FALSE**

**len = 488; prev = 65036;free = TRUE**

**realloc p (65536 bytes)**

**Page #1 attributes:**

**len = 28; prev = 0;free = FALSE**

**len = 65496; prev = 28;free = TRUE**

**Page #2 attributes:**

**len = 65536; prev = 0;free = FALSE**

**free p (65536 bytes)**

**Page #1 attributes:**

**len = 28; prev = 0;free = FALSE**

**len = 65496; prev = 28;free = TRUE**

**Page #2 attributes:**

**len = 65536; prev = 0;free = TRUE**

**free j (84 bytes)**

**Page #1 attributes:**

**len = 65536; prev = 0;free = TRUE**

**Page #2 attributes:**

**len = 65536; prev = 0;free = TRUE**