Міністерство освіти і науки України

Національній технічний університет України "Київський політехнічний інститут"

Факультет прикладної математики

Кафедра спеціалізованих комп’ютерних систем

Лабораторна робота № 2

Дисципліна:

"Системне програмне забезпечення "

Тема:

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

|  |  |
| --- | --- |
| Виконав  Студент групи КВ-92  Сердюк С.О. | Перевірив  Сімоненко В. П. |

**Київ 2012**

**Опис алгоритму**

В реалізації алгоритму “segregated list” пам'ять розбивається на рядки по 6136 бай, де кожен рядок складається із восьми блоків пам’яті різного розміру. Як показано на малюку: 

Кожен 6136-байтний рядок пам’яті містить 8 блоків пам’яті, які зростають в розмірі. Щоб доступитися до блоку пам’яті, потрібно створити для нього індекс та додавати 6136 байт, щоб переміщатися по рядах для даного блоку.  
На початку кожного блоку відведено один байт пам’яті, який показує, зайнятий він чи вільний.

**Часові характеристики**

Для виділення пам’яті затрачується час, прямо пропорційний кількості рядів, тобто O(n). Вивільнення пам’яті відбувається за константний проміжок часу. Для переміщення заданого блока пам’яті буде затрачений час на пошуки даного блока ~ O(n), нове виділення пам’яті, та копіювання даних в новий блок.

**Оцінка розходу пам’яті**

Додаткова пам'ять розходжується по 8 байт для кожного блоку.

**Переваги та недоліки**

Переваги: малий розхід пам’яті для службових структур, простота реалізації, швидке виділення та вивільнення.

Недоліки: малий обмеженість у розмірі максимальної кількості пам’яті, яка може бути виділена за один раз.

**Текст програми**

*File “MemAlloc.h”:*

#pragma once

#define ulong unsigned long

#define uchar unsigned char

#define SZ\_ROW 6136

#define FREE 0

#define OCCUPIED 1

#define START16 1

#define START32 18

#define START64 51

#define START128 116

#define START256 245

#define START512 502

#define START1024 1015

#define START4096 2040

#define STATE(i) (\*((ulong\*)(&ram[i - 1])))

class MemAlloc

{

uchar \* ram;

ulong size;

ulong nrows;

public:

MemAlloc(ulong nbytes);

void initColumn(ulong index);

~MemAlloc(void);

void \* allocate(ulong nbytes);

ulong searchColumn(ulong index);

void release(void \* addr);

void \* realloc(void \* oldAddr, ulong newSize);

void printState();

void printColumn(ulong index);

ulong checkPtr(ulong, void \*);

void \* checkBlock(ulong blockIndex, void \* oldPointer, ulong newSize);

};

*File “MemAlloc.cpp”:*

#include "MemAlloc.h"

#include <stdlib.h>

#include <iostream>

using namespace std;

MemAlloc::MemAlloc(ulong nbytes)

{

ram = (uchar\*)malloc(nbytes);

if (ram)

{

size = nbytes;

if(size < SZ\_ROW)

{

printf("MemoryManager::");

printf("MemoryManager():");

printf("not enough memory fed to constructor\n");

exit(1);

}

nrows = size/SZ\_ROW;

initColumn(START16);

initColumn(START32);

initColumn(START64);

initColumn(START128);

initColumn(START256);

initColumn(START512);

initColumn(START1024);

initColumn(START4096);

}

}

MemAlloc::~MemAlloc(void)

{

free(ram);

}

void MemAlloc::initColumn(ulong index)

{

ulong i;

for(i=0; i < nrows; i++)

{

STATE(index) = FREE;

index = index + SZ\_ROW;

}

return;

}

void \* MemAlloc::allocate(ulong nbytes)

{

ulong index;

if(nbytes <= 16)

{

index = searchColumn(START16);

if(index){ return((void\*)&ram[index]); }

index = searchColumn(START32);

if(index){ return((void\*)&ram[index]); }

index = searchColumn(START64);

if(index){ return((void\*)&ram[index]); }

index = searchColumn(START128);

if(index){ return((void\*)&ram[index]); }

index = searchColumn(START256);

if(index){ return((void\*)&ram[index]); }

index = searchColumn(START512);

if(index){ return((void\*)&ram[index]); }

index = searchColumn(START1024);

if(index){ return((void\*)&ram[index]); }

index = searchColumn(START4096);

if(index){ return((void\*)&ram[index]); }

}

else if(nbytes<=32)

{

index = searchColumn(START32);

if(index){ return((void\*)&ram[index]); }

index = searchColumn(START64);

if(index){ return((void\*)&ram[index]); }

index = searchColumn(START128);

if(index){ return((void\*)&ram[index]); }

index = searchColumn(START256);

if(index){ return((void\*)&ram[index]); }

index = searchColumn(START512);

if(index){ return((void\*)&ram[index]); }

index = searchColumn(START1024);

if(index){ return((void\*)&ram[index]); }

index = searchColumn(START4096);

if(index){ return((void\*)&ram[index]); }

}

else if(nbytes<=64)

{

index = searchColumn(START64);

if(index){ return((void\*)&ram[index]); }

index = searchColumn(START128);

if(index){ return((void\*)&ram[index]); }

index = searchColumn(START256);

if(index){ return((void\*)&ram[index]); }

index = searchColumn(START512);

if(index){ return((void\*)&ram[index]); }

index = searchColumn(START1024);

if(index){ return((void\*)&ram[index]); }

index = searchColumn(START4096);

if(index){ return((void\*)&ram[index]); }

}

else if(nbytes<=128)

{

index = searchColumn(START128);

if(index){ return((void\*)&ram[index]); }

index = searchColumn(START256);

if(index){ return((void\*)&ram[index]); }

index = searchColumn(START512);

if(index){ return((void\*)&ram[index]); }

index = searchColumn(START1024);

if(index){ return((void\*)&ram[index]); }

index = searchColumn(START4096);

if(index){ return((void\*)&ram[index]); }

}

else if(nbytes<=256)

{

index = searchColumn(START256);

if(index){ return((void\*)&ram[index]); }

index = searchColumn(START512);

if(index){ return((void\*)&ram[index]); }

index = searchColumn(START1024);

if(index){ return((void\*)&ram[index]); }

index = searchColumn(START4096);

if(index){ return((void\*)&ram[index]); }

}

else if(nbytes<=512)

{

index = searchColumn(START512);

if(index){ return((void\*)&ram[index]); }

index = searchColumn(START1024);

if(index){ return((void\*)&ram[index]); }

index = searchColumn(START4096);

if(index){ return((void\*)&ram[index]); }

}

else if(nbytes<=1024)

{

index = searchColumn(START1024);

if(index){ return((void\*)&ram[index]); }

index = searchColumn(START4096);

if(index){ return((void\*)&ram[index]); }

}

else if(nbytes<=4096)

{

index = searchColumn(START4096);

if(index){ return((void\*)&ram[index]); }

}

return(NULL);

}//end allocation

ulong MemAlloc::searchColumn(ulong index)

{

ulong i;

for(i = 0; i < nrows; i++)

{

if(STATE(index)==FREE)

{

printf("MemoryManager");

printf("searchColumn(): free at index %lu, ", index);

printf("address=%p\n", &ram[index]);

STATE(index)=OCCUPIED;

return(index);

}

index = index + SZ\_ROW;

}

return(0);

}

void MemAlloc::release(void \* addr)

{

ulong free;

if(addr==NULL)

{

printf("MemoryManager::");

printf("release(): cannot release NULL pointer\n");

return;

}

if( (addr >= (void\*)&ram[size]) || (addr < (void\*)&ram[0]) )

{

printf("MemoryManager::");

printf("release(): address out of bounds\n");

return;

}

free = (ulong)( ((uchar\*)addr) - &ram[0] );

if(free==0)

{

printf("MemoryManager::");

printf("release(): address in first 1st byte\n");

return;

}

if(STATE(free)!=OCCUPIED)

{

printf("MemoryManager::");

printf("release(): referencing invalid region\n");

return;

}

STATE(free) = FREE;

return;

}

void MemAlloc::printState()

{

printf("[16 bytes]");

printColumn(START16);

printf("[32 bytes]");

printColumn(START32);

printf("[64 bytes]");

printColumn(START64);

printf("[128 bytes]");

printColumn(START128);

printf("[256 bytes]");

printColumn(START256);

printf("[512 bytes]");

printColumn(START512);

printf("[1024 bytes]");

printColumn(START1024);

printf("[4096 bytes]");

printColumn(START4096);

return;

}

void MemAlloc::printColumn(ulong index)

{

ulong i;

for(i = 0; i < nrows; i++)

{

if(STATE(index) == OCCUPIED)

{

printf("[%p] ", &ram[index]);

}

index = index + SZ\_ROW; }

printf("\n");

return;

}

ulong MemAlloc::checkPtr(ulong index, void \* ptr)

{

for(ulong i = 0; i < nrows; i++)

{

if (ptr == (void \*)&ram[index])

if (STATE(index) == OCCUPIED)

return index;

index = index + SZ\_ROW;

}

return 0;

}

void \* MemAlloc::checkBlock(ulong blockIndex, void \* oldPointer, ulong newSize)

{

void \* newPointer;

ulong index = checkPtr(blockIndex, oldPointer);

if (index)//пошук блока, на який дано вказівник

{

STATE(index) = FREE;

newPointer = allocate(newSize);

if (newPointer != oldPointer)

for (ulong i = 0; i < newSize; i++)

((char \*)newPointer)[i] = ((char \*)oldPointer)[i];

return newPointer;

}

return 0;

}

void \* MemAlloc::realloc(void \* oldPointer, ulong newSize)

{

void \* newPointer;

if (newPointer = checkBlock(START16, oldPointer, newSize)) {return newPointer;}

if (newPointer = checkBlock(START32, oldPointer, newSize)) {return newPointer;}

if (newPointer = checkBlock(START64, oldPointer, newSize)) {return newPointer;}

if (newPointer = checkBlock(START128, oldPointer, newSize)) {return newPointer;}

if (newPointer = checkBlock(START256, oldPointer, newSize)) {return newPointer;}

if (newPointer = checkBlock(START512, oldPointer, newSize)) {return newPointer;}

if (newPointer = checkBlock(START1024, oldPointer, newSize)) {return newPointer;}

if (newPointer = checkBlock(START4096, oldPointer, newSize)) {return newPointer;}

return 0;

}

*File “MemoryMannager.h”:*

#pragma once

void initMemMgr(unsigned long totalbytes);

void closeMemMgr();

void \* mem\_alloc(unsigned long size);

void mem\_free(void \*ptr);

void \* mem\_realloc(void \* oldPtr, unsigned long newSize);

void mem\_dump(void);

*File “MemoryManager.cpp”:*

#include "MemAlloc.h"

MemAlloc \*mmptr;

void initMemMgr(unsigned long totalbytes)

{

mmptr = new MemAlloc(totalbytes);

}

void closeMemMgr()

{

delete(mmptr);

}

void \* mem\_alloc(unsigned long size)

{

void \*ptr = mmptr->allocate(size);

return(ptr);

}

void mem\_free(void \*ptr){

mmptr->release(ptr);

return;

}

void mem\_dump(){

mmptr->printState();

}

*File “Test.cpp”:*

#include "MemoryManager.h"

#include <stdio.h>

void debugTest()

{

void \*ptr[6];

unsigned long allocs[6]={8,12,33,1,122,50};

int i;

initMemMgr(20000);

for(i=0;i<6;i++)

{

ptr[i] = mem\_alloc(allocs[i]);

if(ptr[i]==NULL){ printf("ptr[%lu]==NULL!\n",i); }

}

printf("\n\MEMORY DUMP------------------------------\n\n");

mem\_dump();

printf("\n\nFREE MEMORY------------------------------\n\n");

mem\_free(ptr[0]); //8

mem\_free(ptr[3]); //1

mem\_free(ptr[4]); //122

mem\_free(ptr[2]); //33

mem\_free(ptr[1]); //12

mem\_free(ptr[5]); //50

closeMemMgr();

return;

}

void main()

{ debugTest();

return;

}

***Результат роботи***  
