Лабораторная робота №2

По курсу «Системное программное обеспечение» на тему:

«Аллокатор памяти общего назначения(Часть 2)»

Выполнил Проверил(-ла)

Студент группы КВ-82 Симоненко В.П.

Бабенко Ярослав Дробязко И.П.

Постановка задачи

Разработать аллокатор общего назначения, используя за основу описанный выше базовый вариант алгоритма, принимая во внимания следующие условия:

1. Области памяти можно выделять любым доступным способом.
2. Функции mem\_alloc(), mem\_realloc() и mem\_free() должны соответствовать приведенным выше прототипам.
3. Адреса памяти, возвращаемые функциями mem\_alloc() и mem\_realloc(), должны быть выровнены на границу в 4 байта.
4. Попытаться уменьшить время поиска свободного блока памяти и время освобождения занятого блока.
5. Попытаться уменьшить фрагментацию памяти.
6. Написать функцию mem\_dump(), которая должна выводить на консоль состояние областей памяти.
7. Описание разработанного алгоритма.

В версии аллокатора с применением одного списка свободных блоков мы тратили много ресурсов на разделение и слияния блоков памяти. Новый подход с использование нескольких списков пытается обойти эту проблему, сохраняя несколько списков с фиксированными блоками памяти. Другими словами хранение "кучи" осуществляется с помощью группы блоков, основанных на их размере. Если Вы нуждаетесь в 32-байтовом блоке, запрашиваете соответствующий список для свободного пространства вместо того, чтобы пересечь всю "кучу".

В моей отдельной реализации списка я размещаю память в 6 136-байтовые страницы, где каждая строка состоит из восьми различных блоков памяти. Размер первого элемента в каждой строке составляет 16 байтов, и размер последнего элемента в каждой строке составляет 4 096 байтов.

1. Оценка времени поиска свободного блока памяти, оценка времени освобождения занятого блока.

В худшем случае придется проверить n заголовков списка. Освобождение проходит очень быстро достаточно только поменять значение заголовка с 0 на 1.

1. Оценка расхода памяти для хранения служебной информации.

Каждый блок памяти включает заголовок размером 1 байт.

1. Описание достоинств и недостатков разработанного аллокатора.

К достоинствам можно отнести большую скорость работы, главный недостаток это детерминированность аллокатора так как разделение на блоки проходит при инициализации аллокатора.

1. Листинг аллокатора памяти общего назначения.

#include<windows.h>

#include<stdio.h>

#include<stdlib.h>

#include "memmgr.h"

#define STATE(i) (\*((unsigned char\*)(&ram[i-1]))) /\*FREE,OCCUPIED\*/

#define FREE 0

#define OCCUPIED 1

char \*stateStr[3]={"FREE","OCCUPIED"};

// Constructor.

SegregatedMemoryManager::SegregatedMemoryManager(unsigned long nbytes) {

handle = GetProcessHeap();

if (handle == NULL) {

printf("SegregatedMemoryManager::");

printf("SegregatedMemoryManager():");

printf("invalid handle\n");

exit(1);

}

ram = (unsigned char \*)HeapAlloc(handle, HEAP\_ZERO\_MEMORY, nbytes);

size = nbytes;

if (size < SZ\_ROW) {

printf("SegregatedMemoryManager::");

printf("SegregatedMemoryManager():");

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);

printf("SegregatedMemoryManager::");

printf("SegregatedMemoryManager(%lu)\n", nbytes);

return;

}

// Init column.

void SegregatedMemoryManager::InitColumn(unsigned long index) {

unsigned long i;

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

STATE(index) = FREE;

index = index + SZ\_ROW;

}

return;

}

// Destructor.

SegregatedMemoryManager::~SegregatedMemoryManager() {

if (HeapFree(handle, HEAP\_NO\_SERIALIZE, ram) == 0) {

printf("SegregatedMemoryManager::");

printf("~SegregatedMemoryManager():");

printf("could not free heap storage\n");

return;

}

printf("SegregatedMemoryManager::");

printf("~SegregatedFitMemoryManager()");

printf("free ram[%lu]\n", size);

return;

}

// nbytes - number of bytes required

// returns address of first byte of memory region allocated

// (or NULL if cannot allocate a large enough block).

void \*SegregatedMemoryManager::Allocate(unsigned long nbytes) {

unsigned long index;

printf("SegregatedMemoryManager::");

printf("allocate(%lu)\n", nbytes);

if (nbytes==0) {

printf("SegregatedMemoryManager::");

printf("allocate(): zero bytes requested\n");

return NULL;

}

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;

}

// Search a given size range for a free element

// return index

// or zero if no available memory.

unsigned long SegregatedMemoryManager::SearchColumn(unsigned long index) {

unsigned long i;

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

if (STATE(index) == FREE) {

printf("SegregatedMemoryManager::");

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

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

STATE(index) = OCCUPIED;

return index;

}

index = index + SZ\_ROW;

}

return 0;

}

// Release the memory.

void SegregatedMemoryManager::Release(void \*addr)

{

unsigned long free; //index into ram[]

if (addr == NULL) {

printf("SegregatedMemoryManager::");

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

return;

}

printf("SegregatedMemoryManager::");

printf("release(%p)\n", addr);

// Perform sanity check to make sure address is kosher.

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

printf("SegregatedMemoryManager::");

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

return;

}

// Translate void\* addr to index in ram[].

free = (unsigned long)(((unsigned char \*)addr) - &ram[0]);

printf("SegregatedMemoryManager::");

printf("address resolves to index %lu\n",free);

// Header always occupies first 13 bytes of storage.

if (free == 0) {

printf("SegregatedMemoryManager::");

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

return;

}

// Yet more sanity checks.

if (STATE(free) != OCCUPIED) {

printf("SegregatedMemoryManager::");

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

return;

}

STATE(free) = FREE;

return;

}

void SegregatedMemoryManager::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 SegregatedMemoryManager::PrintColumn(unsigned long index) {

unsigned long i;

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

if(STATE(index) == OCCUPIED) {

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

}

index = index + SZ\_ROW;

}

printf("\n");

return;

}

