Лабораторная робота №1

По курсу «Системное программное обеспечение» на тему:

«Аллокатор памяти общего назначения»

Выполнил Проверил

Студент группы КВ-81 Симоненко В.П.

Сухомлин Александр

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.

Функция mem\_alloc будет запрашивать память у операционной системы по мере возможности. Поскольку в ходе других операций в программе также может запрашиваться память без помощи этой функции распределения памяти, управляемое mem\_alloc пространство может оказаться не непрерывным. Таким образом, свободная память будет храниться в виде списка свободных блоков.

Каждый блок содержит свой размер, указатель на следующий блок и сам участок памяти. Блоки хранятся в порядке возрастания адресов, а последний блок (с самым старшим адресом) ссылается на первый.

Когда поступает запрос на память, выполняется перебор списка свободных блоков, пока не обнаруживается достаточно большой блок.

Если блок имеет точно такой размер, как запрашивается, он отсоединяется от списка и передается пользователю. Если блок имеет больший размер, он делится на части, и пользователь получает столько, сколько просил, а остаток остается в списке. Если не удается найти достаточно большой блок, у операционной системы запрашивается очередной большой фрагмент памяти,

который подключается к списку.

При освобождении также выполняется поиск по списку свободных блоков:

отыскивается место для вставки освобождаемого блока. Если освобождаемый блок вплотную граничит со свободным блоком с какой-либо из двух сторон, то он сливается с ним в единый блок большего размера, чтобы не слишком фрагментировать память. Определить, как граничат блоки, нетрудно, поскольку список свободных блоков организован по возрастанию адресов.

Свободный блок содержит указатель на следующий блок в цепочке, размер блока и собственно участок свободной памяти.

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

В худшем случае при поиске свободного блока потребуется проверить N заголовков. Освобождение блока происходит быстро, для этого достаточно проверить заголовки левого и правого соседних блоков

1. Оценка расхода памяти для хранения служебной информации.

Для N блоков потребуется (N\*8) байт дополнительной информации. Также потребуется память для хранения указателей на области памяти.

1. Описание достоинств и недостатков разработанного аллокатора.

Адреса памяти выровнены на границу в 4 байта, что снижает размер затрачиваемой памяти при выделении очень маленьких кусков памяти. Также быстро происходит освобождение памяти. Недостаток заключается в том, что когда поступает запрос на память, выполняется перебор списка свободных блоков, пока не обнаруживается достаточно большой блок, то есть первый подходящий блок а не наилучший, через что выделяемая память может быть очень фрагментирована.

1. Листинг аллокатора памяти общего назначения.

#include "stdio.h"

#include "windows.h"

#include "string.h"

#define NALLOC 1024

typedef long align;

struct header {

struct header \*ptr;

size\_t size;

};

typedef struct header Header;

Header base; // Empty list to get started.

Header \*free\_ptr = NULL; // Start of the free list.

void mem\_free(void \*addr) {

Header \*bp;

Header \*p;

bp = (Header \*)addr - 1; // Pointer on header.

for (p = free\_ptr; !(bp > p && bp < p->ptr); p = p->ptr) { // Freed block at start or end of arena.

if (p >= p->ptr && (bp > p || bp < p->ptr))

break;

}

if ((unsigned long \*)bp + bp->size == p->ptr) { // To high adjacent block.

bp->size += p->ptr->size;

bp->ptr = p->ptr->ptr;

} else {

bp->ptr = p->ptr;

}

if ((unsigned long \*)p + p->size == bp) { // To down adjacent block.

p->size += bp->size;

p->ptr = bp->ptr;

} else {

p->ptr = bp;

}

free\_ptr = p;

}

Header \*call\_memory(size\_t n\_blocks) {

HANDLE handle;

unsigned char \*ram\_ptr; // Memory storage.

Header \*new\_memory;

handle = GetProcessHeap();

if (handle == NULL) {

printf("Invalid handle.\n");

exit(1);

}

if (n\_blocks < NALLOC) {

n\_blocks = NALLOC;

}

ram\_ptr = (unsigned char \*)HeapAlloc(handle, HEAP\_ZERO\_MEMORY, n\_blocks \* sizeof(unsigned long));

new\_memory = (Header \*)ram\_ptr;

new\_memory->size = n\_blocks;

mem\_free((void \*)(new\_memory + 1));

return free\_ptr;

}

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

Header \*p, \*prev\_ptr;

size\_t n\_blocks;

n\_blocks = (size + sizeof(unsigned long) - 1) / sizeof(unsigned long) + 2;

if ((prev\_ptr = free\_ptr) == NULL ) { // List is empty.

prev\_ptr = &base;

free\_ptr = prev\_ptr;

base.ptr = free\_ptr;

base.size = 0;

}

for (p = prev\_ptr->ptr; ;prev\_ptr = p, p = p->ptr) {

if (p->size >= n\_blocks) {

if (p->size == n\_blocks) {

prev\_ptr->ptr = p->ptr;

} else {

p->size -= n\_blocks;

(unsigned long \*)p += p->size;

p->size = n\_blocks;

}

free\_ptr = prev\_ptr;

return (void \*)(p + 1);

}

if (p == free\_ptr) {

if ((p = call\_memory(n\_blocks)) == NULL) {

return NULL;

}

}

}

}

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

unsigned char \*new\_addr;

size\_t old\_n\_blocks = 0;

size\_t new\_n\_blocks = 0;

Header \*bp;

if (addr == NULL) { return mem\_alloc(size); }

if (size == old\_n\_blocks\* sizeof(unsigned long)) { return addr; }

new\_addr = mem\_alloc(size);

bp = (Header \*)addr - 1; // Pointer on header.

old\_n\_blocks = bp->size;

new\_n\_blocks = (size + sizeof(unsigned long) - 1) / sizeof(unsigned long) + 2; // +2 blocks by 4 bytes for header.

if (old\_n\_blocks > new\_n\_blocks)

memcpy(new\_addr, addr, (new\_n\_blocks - 2) \* sizeof(unsigned long));

else

memcpy(new\_addr, addr, (old\_n\_blocks - 2) \* sizeof(unsigned long));

mem\_free(addr);

return (void \*)new\_addr;

}

void mem\_dump() {

Header \*p = &base;

do {

printf("[ addr=%p] [size=%4.1d] [next=%p]\n", p, p->size, p->ptr);

p = p->ptr;

} while (p != &base);

}

int main() {

void \*ptr1, \*ptr2, \*ptr3, \*ptr4;

ptr1 = mem\_alloc(2000);

ptr2 = mem\_alloc(2000);

ptr3 = mem\_alloc(4000);

ptr4 = mem\_alloc(105);

mem\_dump();

printf("MEM\_ALLOC\n");

printf("[ptr=%p] [size=%d]\n", ptr1, ((Header \*)ptr1 - 1)->size);

printf("[ptr=%p] [size=%d]\n", ptr2, ((Header \*)ptr2 - 1)->size);

printf("[ptr=%p] [size=%d]\n", ptr3, ((Header \*)ptr3 - 1)->size);

printf("[ptr=%p] [size=%d]\n", ptr4, ((Header \*)ptr4 - 1)->size);

printf("MEM\_REALLOC\n");

ptr1 = mem\_realloc(ptr1, 100);

ptr2 = mem\_realloc(ptr2, 1000);

ptr3 = mem\_realloc(ptr3, 8000);

ptr4 = mem\_realloc(ptr4, 5);

printf("[ptr=%p] [size=%d]\n", ptr1, ((Header \*)ptr1 - 1)->size);

printf("[ptr=%p] [size=%d]\n", ptr2, ((Header \*)ptr2 - 1)->size);

printf("[ptr=%p] [size=%d]\n", ptr3, ((Header \*)ptr3 - 1)->size);

printf("[ptr=%p] [size=%d]\n", ptr4, ((Header \*)ptr4 - 1)->size);

printf("MEM\_FREE\n");

mem\_free(ptr4);

mem\_free(ptr3);

mem\_free(ptr2);

mem\_free(ptr1);

mem\_dump();

return 0;

}

1. Пример работы аллокатора.

