Лабораторна робота №1

По курсу «Системне програмне забезпечення»

на тему:

«Алокатор пам’яті загального призначення»

Виконав:

студент групи КВ-92

Степанюк Михайло Федорович

Перевірив:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Постановка задачі

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, які повертаються функціями mem\_alloc() і mem\_realloc(), повинні бути вирівняні до межі в 4 байти.
4. Спробувати зменшити час пошуку вільного блоку пам’яті і час звільнення зайнятого блоку.
5. Спробувати зменшити фрагментацію пам’яті.
6. Написати функцію mem\_dump(), яка повинна виводити на консоль стан областей пам’яті.
7. Описання розробленого алгоритму.

Функція mem\_alloc буде запрошувати пам’ять у операційної системи по мірі можливості. Оскільки в ході інших операцій в програмі також може запрошуватись пам’ять без допомоги цієї функції і розподілу пам’яті, керований mem\_alloc простір може виявитися не неперервним. Таким чином, вільна пам’ять буде зберігатися у вигляді списку вільних блоків.

Кожен блок містить свій розмір, вказівник на наступний блок і саму ділянку памяті. Блоки зберігаються в порядку зростання адрес, а останній блок (з найстаршою адресою) засилається на перший.

Коли поступає запит на пам’ять, виконується перебір списку вільних блоків, доки не буде знайдено достатньо великий блок.

Якщо блок має точно такий же розмір, як і той, що запрошується, він від’єднується від списку і передається користувачу. Якщо блок має більший розмір, він ділиться на частини, а користувач отримує стільки, скільки просив, а залишок залишається в списку. Якщо не вдається знайти достатньо великий блок, у операційної системи запрошується черговий великий фрагмент пам’яті, який підключається до списку.

При звільненні також виконується пошук по списку вільних блоків:

- відшукується місце для вставки звільняємого блока.

Якщо звільняємий блок критично межує з вільним блоком з якогось з двог боків, то він зливається з ним в єдиний блок великого розміру, щоб не надто фрагментувати пам’ять. Визначити як межують блоки, нетяжко, оскільки список вільних блоків організований по зростанню адрес.

Вільний блок містить вказівник на наступний блок в ланцюжку, розмір блоку і власне ділянку вільної пам’яті.

1. Оцінка часу пошуку вільного блоку пам’яті, оцінка часу звільнення зайнятого блоку.

В найгіршому випадку при пошуку вільного блоку потрібно перевірити N заголовків. Звільнення блоку відбувається швидко, для цього достатньо перевірити заголовки лівого та правого сусідніх блоків

1. Оцінка використання пам’яті для зберігання службової інформації.

Для N блоків потрібно (N\*8) байт додаткової інформації. Також потрібна буде пам’ять для збереження вказівників на області пам’яті.

1. Описання переваг і недоліків розробленого алокатора.

Адреси пам’яті вирівняні на межу в 4 байти, що знижує розмір затрачаємої пам’яті при виділенні дуже малих шматків пам’яті. Також швидко відбувається звільнення пам’яті. Недолік заключається в тому, що коли поступає запит на пам’ять, виконується перебір списку вільних блоків, доки не буде знайдено достатньо великий блок, тобто перший підходящий блок, а не найкращий, через що виділяєма пам’ять може бути дуже фрагментована.

1. Лістинг алокатора пам’яті загального призначення.

#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. Приклад роботи алокатора.

