Міністерство освіти і науки України

Національний технічний університет України

«Київський Політехнічний Інститут»

Факультет прикладної математики

Кафедра Спеціалізованих комп’ютерних систем

Лабораторна робота №2

З дисципліни «Системне програмне забезпечення» :

«Алокатор пам’яті загального призначення ( ч.2 )»

Варіант №16

Виконав:

студент групи КВ-91

Федай Г. Г.

Перевірив:

Симоненко В. П.

Київ 2012

**Опис розробленого алгоритму**

Для розробки алокатору було застосовано алгоритм близнюків. Заданий масив списків, кожен з яких відповідає певній області пам’яті, ємність якої дорівнює степені двійки. При виділення пам’яті об’єм пам’яті округлюється до найближчого числа степені двійки і пошук вільного блоку відбувається в тому із списків, які містять інформацію про всі блоки заданого розміру. Якщо цей список порожній, ми беремо список блоків вдвічі більшого розміру. Отримавши блок удвічі більшого розміру, ми ділимо його навпіл. Непотрібну половину ми поміщаємо у відповідний список вільних блоків. При звільненні пам’яті сусідні блоки зчіплюються за умови, якщо вони вільні і різниця між зсувами адрес дорівнює ємності заданого блоку. В результаті отримуємо блок пам’яті вдвічі більшої ємності і поміщаємо його до відповідного списку.

Даний алгоритм дає переваги у швидкому пошуку необхідного вільного блоку пам’яті, зменшенні фрагментації.

Лістинг програми:

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

int lim = 1024,size\_mem = 1024, num = 9;

int mem\_mas[] = {4,8,16,32,64, 128, 256, 512, 1024};

bool\* mem;

struct headline

{

bool state;

int free\_sz;

int all\_len;

int addr;

headline \*next;

headline \*prev;

};

struct headline \*main\_mem;

struct headline\* queue[9];

void InitHead ( headline \*head )

{

head->all\_len = size\_mem;

head->free\_sz = size\_mem;

head->state = false;

head->addr = 0;

head->next = NULL;

head->prev = NULL;

}

void InitQueue ( )

{

int i, rest\_mem = size\_mem;

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

queue[i] = NULL;

headline \*beg = NULL, \*node;

node = (headline \*)malloc(sizeof(headline));

node->addr = 0;

node->all\_len = size\_mem;

node->free\_sz = node->all\_len;

node->state = false;

node->next = NULL;

node->prev = NULL;

queue[num-1] = node;

return;

}

void CopyData(headline \*el\_to, headline el\_from )

{

el\_to->addr = el\_from.addr;

el\_to->free\_sz = el\_from.free\_sz;

el\_to->all\_len = el\_from.all\_len;

el\_to->state = el\_from.state;

return;

}

void PrintQu()

{

int i = 0;

headline \*ptr = NULL;

for ( int i = 0; i<num; i++ )

{

ptr = queue[i];

printf("\n\t\t mem\_size: %d\n", mem\_mas[i]);

while ( ptr != NULL )

{

printf("address %d full\_size %d free\_size %d state %d\n", ptr->addr, ptr->all\_len, ptr->free\_sz,ptr->state);

ptr = ptr->next;

}

}

return;

}

headline\* add\_el ( headline buf, int pos )

{

headline \*ptr = queue[pos], \*ins\_el;

ins\_el = (headline \*)malloc(sizeof(headline));

CopyData(ins\_el,buf);

if ( pos < 0 )

return NULL;

if ( queue[pos] == NULL )

{

ins\_el->next = NULL;

ins\_el->prev = NULL;

queue[pos] = ins\_el;

return ins\_el;

}

while ( ptr != NULL && ptr->addr < ins\_el->addr )

{

ptr = ptr->next;

}

if ( ptr == NULL )

{

ptr = queue[pos];

while ( ptr->next != NULL )

ptr = ptr->next;

ptr->next = ins\_el;

ins\_el->prev = ptr;

ins\_el->next = NULL;

return ins\_el;

}

ins\_el->next = ptr;

ins\_el->prev = ptr->prev;

if ( ptr->prev != NULL )

ptr->prev->next = ins\_el;

else

queue[pos] = ins\_el;

ptr->prev = ins\_el;

return ins\_el;

}

int alloc\_fun ( size\_t size, int pos )

{

int full\_sz = mem\_mas[pos];

headline \*buf, \*ptr, \*free\_el;

buf = (headline \*)malloc(sizeof(headline));

free\_el = (headline \*)malloc(sizeof(headline));

buf->state = false;

while ( pos < num )

{

ptr = queue[pos];

while ( ptr != NULL && ptr->state == true )

ptr = ptr->next;

if ( ptr == NULL )

pos++;

else

{

full\_sz = mem\_mas[pos];

break;

}

}

while ( ptr!= NULL && full\_sz > 4 && size <= full\_sz/2 )

{

free\_el->all\_len = full\_sz/2;

free\_el->free\_sz = free\_el->all\_len;

free\_el->state = false;

free\_el->addr = ptr->addr + free\_el->all\_len;

add\_el(\*free\_el, pos-1);

ptr->all\_len = full\_sz/2;

ptr->free\_sz = ptr->all\_len - size;

ptr->state = true;

if ( ptr->prev == NULL )

queue[pos] = ptr->next;

else

ptr->prev->next = ptr->next;

if ( ptr->next != NULL )

ptr->next->prev = ptr->prev;

ptr = add\_el(\*ptr, pos-1);

pos--;

full\_sz /=2;

}

if ( ptr == NULL )

return -1;

ptr->free\_sz = ptr->all\_len - size;

ptr->state = true;

return ptr->addr;

}

int mem\_alloc( size\_t size )

{

int i = 0, full\_sz = 0;

while ( i < num && size > mem\_mas[i] )

i++;

if ( i >= num )

return -1;

full\_sz = mem\_mas[i];

headline \*node = queue[i];

while ( node == NULL )

{

full\_sz\*=2;

node = queue[++i];

}

if ( full\_sz > lim )

return -1;

return alloc\_fun(size,i);

}

int mem\_free( int \_addr )

{

int i = 0, pos = 0;

headline \*ptr = NULL, \*ptr2 = NULL;

while ( i < num )

{

ptr = queue[i];

while ( ptr != NULL )

{

if ( ptr->addr == \_addr )

{

pos = i;

i = num;

break;

}else

ptr = ptr->next;

}

i++;

}

ptr->state = false;

ptr->free\_sz = ptr->all\_len;

i = pos;

while ( i < num-1 )

{

if ( ptr->next != NULL && ptr->next->state == false && (ptr->next->addr - ptr->addr ) == mem\_mas[i])

ptr2 = ptr->next;

else

if ( ptr->prev != NULL && ptr->prev->state == false && (ptr->addr - ptr->prev->addr ) == mem\_mas[i])

{

ptr2 = ptr;

ptr = ptr->prev;

}else

break;

ptr->all\_len \*= 2;

ptr->free\_sz = ptr->all\_len;

ptr->next = ptr2->next;

if ( ptr2->next != NULL )

ptr2->next->prev = ptr;

ptr2->next = NULL;

ptr2->prev = NULL;

ptr2 = NULL;

if ( ptr->prev == NULL )

queue[i] = ptr->next;

else

ptr->prev->next = ptr->next;

if ( ptr->next != NULL )

ptr->next->prev = ptr->prev;

ptr->next = NULL;

ptr->prev = NULL;

ptr = add\_el(\*ptr, i+1);

i++;

}

return ptr->addr;

}

int main ()

{

InitQueue();

PrintQu();

printf("\n");

mem\_alloc(64);

mem\_alloc(64);

mem\_alloc(64);

PrintQu();

mem\_free(0);

mem\_free(64);

//mem\_free(128);

PrintQu();

return 0;

}