

Міністерство освіти і науки України

Національний технічний університет України

«Київський політехнічний інститут»

**Лабораторна робота №3**

***з дисципліни «Введення до операційних систем»***

**«Керування пам’яттю»**

Виконав студент групи: КВ-11

ПІБ: Терентьєв Іван Дмитрович

Перевірив: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Київ 2024**

*Загальне завдання*

1. Написатипрограму, що моделює процес управлінняпам’яттю (розподілу

пам’яті для процесів), перетворення віртуальної адреси у фізичну, пошук у пам’яті за запитами процесів, вивільнення пам’яті) при заданому варіантом способі організації пам’яті (перелік варіантів представлений нижче).

Вхідні дані– розмір пам’яті, що підлягає розподілу, розміри сторінок (розділів, сегментів тощо), розміри потрібної процесам пам’яті та ін. задаються самостійно та у відповідності до завдання.

1. Продемонструвати роботу моделі з виконанням основних операцій з пам’яттю: надання пам’яті потрібного розміру за запитом процесу, перетворення віртуальної адреси у “фізичну” при зверненні до комірки пам’яті, здійнення запису або читання, вивільнення пам’яті при завершенні процесу. Завдання операцій можна реалізувати за допомогою меню.

Якщо потрібно за алгоритмом, додатково продемонструвати процес вивантаження-завантаження сегментів або сторінок.

Вихідні дані – нагляднаінформація про поточний розподілпам’яті (карта пам’яті), що містить адреси ділянок пам’яті, стан (вільно чи зайнято та ким) після кожної операції з пам’яттю.

Окремо показати коректність перетворення віртуальної адреси звернення до комірки пам’яті за запитом у “фізичну” адресу і здійнення запису до комірки та читання з неї.

*Індивідуальне завдання за варіантом 23(8)*

Переміщувальні розділи (без використання зовнішньої пам’яті). Кількість розділів – меньша, ніж кількість процесів. Якщо черговий розділ неможливо розмістити у пам’яті, виконується процедура «стискування» в напрямку молодших адрес. Процеси утворюють загальну чергу до розділів пам’яті. Використовується сегментований адресний простір. Сегменти можуть розміщуватися в різних розділах. Розміри процесів задаються випадково.

*Код програми:*

==> main.c <==

#**include** <stdio.h>

#**include** <stdlib.h>

#**include** <time.h>

#**include** <math.h>

#**include** <stdbool.h>

#**include** <string.h>

#**include** "config.h"

#**include** "memory\_controller.h"

#**include** "physical\_memory.h"

size\_t current\_proc = 0;

void **print\_phys\_full**(phys\_mem \*phys)

{

printf("Output memory:\n");

printf("RAM = %llu\n", phys->size);

printf("Volume count = %llu\n", phys->volume\_count);

printf("Volume==================================\n");

**for** (size\_t i = 0; i < phys->volume\_count; i++)

{

printf("Volume %llu\n", i);

printf("Base: %llu\nSize: %llu\n", phys->volumes[i].base, phys->volumes[i].size);

printf("Process count: %llu\n", phys->volumes[i].procs\_count);

printf("Process=================================\n");

**for** (size\_t k = 0; k < phys->volumes[i].procs\_count; k++)

{

printf("Process %llu\n", k);

printf("Physical address: ");

phys\_addr proc\_phys = (get\_phys\_addr(phys->volumes[i].procs[k], &phys->volumes[i]));

print\_phys\_addr(&proc\_phys);

printf("\n");

printf("Lifetime: %llu\n", phys->volumes[i].procs[k]->lifetime);

printf("Segments count: %llu\n", phys->volumes[i].procs[k]->segments\_count);

printf("Segment=================================\n");

**for** (size\_t j = 0; j < phys->volumes[i].procs[k]->segments\_count; j++)

{

printf("Segment %llu\n", phys->volumes[i].procs[k]->segments[j].id);

printf("Size: %llu\n", phys->volumes[i].procs[k]->segments[j].size);

printf("Identifiers count: %llu\n", phys->volumes[i].procs[k]->segments[j].idents\_count);

printf("Identifier==============================\n");

printf("Name |Vseg|Voff|Size|Physical addr \n");

printf("########################################\n");

**for** (size\_t a = 0; a < phys->volumes[i].procs[k]->segments[j].idents\_count; a++)

{

phys\_addr ident\_phys = (get\_phys\_addr\_ident(phys->volumes[i].procs[k]->segments[j].idents[a].name,

&phys->volumes[i].procs[k]->segments[j], phys->volumes[i].procs[k], &phys->volumes[i]));

printf("%10s|%4llu|%4llu|%4llu|", phys->volumes[i].procs[k]->segments[j].idents[a].name,

phys->volumes[i].procs[k]->segments[j].idents[a].addr.segment\_id,

phys->volumes[i].procs[k]->segments[j].idents[a].addr.addr\_in\_seg,

phys->volumes[i].procs[k]->segments[j].idents[a].size);

print\_phys\_addr(&ident\_phys);

printf("\n");

}

}

}

}

}

void **load\_proc\_to\_mem**(mem\_control \*mem, phys\_mem \*phys)

{

**if** (mem->procs\_count > 0)

{

proc \*my\_proc = &mem->procs[current\_proc];

**if** (get\_proc\_size(my\_proc) <= mem->free\_memory)

{

bool status = create\_volume(phys, my\_proc);

**if** (status){

printf("Loaded\n");

mem->free\_memory -= get\_proc\_size(my\_proc);

mem->used\_memory += get\_proc\_size(my\_proc);

current\_proc++;

}

**else**

printf("Cannot load this proc\n");

}

**else**

{

printf("Sorry, unload some processses, not enough memory\n");

}

}

**else**

{

printf("Sorry, restart program to generate new procs\n");

}

}

void **load\_proc\_to\_spec\_mem**(mem\_control \*mem, phys\_mem \*phys)

{

printf("Input volume id\n");

size\_t value;

scanf("%llu", &value);

**if** (value < phys->volume\_count)

{

**if** (mem->procs\_count > 0)

{

proc \*my\_proc = &mem->procs[current\_proc];

**if** (get\_proc\_size(my\_proc) <= mem->free\_memory)

{

bool status = add\_to\_volume(phys, my\_proc, value);

**if** (status){

printf("Loaded\n");

mem->free\_memory -= get\_proc\_size(my\_proc);

mem->used\_memory += get\_proc\_size(my\_proc);

current\_proc++;

}

**else**

printf("Cannot load this proc\n");

}

**else**

{

printf("Sorry, unload some processses, not enough memory\n");

}

}

**else**

{

printf("Sorry, restart program to generate new procs\n");

}

}

**else**

{

printf("This volume id does not exist");

}

}

void **proc\_tick**(mem\_control \*mem, phys\_mem \*phys, size\_t tick)

{

**for** (size\_t i = 0; i < phys->volume\_count; i++)

{

**for** (size\_t k = 0; k < phys->volumes[i].procs\_count; k++)

{

phys->volumes[i].procs[k]->lifetime--;

**if** (phys->volumes[i].procs[k]->lifetime == 0){

mem->free\_memory+=get\_proc\_size(phys->volumes[i].procs[k]);

mem->used\_memory-=get\_proc\_size(phys->volumes[i].procs[k]);

delete\_proc\_from\_volume(phys, phys->volumes[i].procs[k], i);

mem->procs\_count--;

**if**(phys->volume\_count == 0)

**break**;

}

}

}

printf("Tick %llu\n", tick);

}

void **unload\_proc**(mem\_control \*mem, phys\_mem \*phys)

{

printf("Enter volume id: ");

size\_t volume\_id = SIZE\_MAX;

scanf("%llu", &volume\_id);

**if** (volume\_id < phys->volume\_count)

{

printf("Enter proc id: ");

size\_t proc\_id = SIZE\_MAX;

scanf("%llu", &proc\_id);

**if** (proc\_id < phys->volumes[volume\_id].procs\_count)

{

mem->free\_memory+=get\_proc\_size(phys->volumes[volume\_id].procs[proc\_id]);

mem->used\_memory-=get\_proc\_size(phys->volumes[volume\_id].procs[proc\_id]);

mem->procs\_count--;

delete\_proc\_from\_volume(phys, phys->volumes[volume\_id].procs[proc\_id], volume\_id);

printf("Success\n");

}

}

}

void **menu**()

{

printf("MENU\n");

printf("a. Print memory\n");

printf("b. Load one random proc to memory\n");

printf("c. Load one random proc to specified volume\n");

printf("d. Processor tick\n");

printf("e. Force unload proc from memory\n");

printf("q. Exit\n");

}

int **main**()

{

srand((unsigned int)time(NULL));

size\_t procs\_count = 10;

proc \*procs = generate\_procs(procs\_count);

mem\_control mem = {.max\_size = RAM\_SIZE,

.procs = procs,

.procs\_count = procs\_count,

.used\_memory = 0,

.free\_memory = RAM\_SIZE};

phys\_mem physical = {.size = RAM\_SIZE,

.volume\_count = 0,

.volumes = NULL};

size\_t cpu\_tick = 0;

char c = '\n';

**do**

{

**switch** (c)

{

**case** 'a':

print\_phys\_full(&physical);

**break**;

**case** 'b':

load\_proc\_to\_mem(&mem, &physical);

**break**;

**case** 'c':

load\_proc\_to\_spec\_mem(&mem, &physical);

**break**;

**case** 'd':

proc\_tick(&mem, &physical, cpu\_tick);

cpu\_tick++;

**break**;

**case** 'e':

unload\_proc(&mem, &physical);

**break**;

**default**:

menu();

**break**;

};

c = (char)getchar();

} **while** (c != 'q');

**return** 0;

}

==> physical\_address.c <==

#**include** "physical\_address.h"

void print\_phys\_addr(phys\_addr\* pa)

{

printf("%2lux%4lu",pa->base,pa->offset);

}

==> physical\_memory.c <==

#**include** "physical\_memory.h"

bool create\_volume(phys\_mem\* memory, proc\* my\_proc)

{

size\_t curr\_offset = 0;

**if**(memory->volume\_count != 0)

curr\_offset = memory->volumes[memory->volume\_count-1].base + memory->volumes[memory->volume\_count-1].size;

**if**(get\_proc\_size(my\_proc)+curr\_offset > memory->size)

{

curr\_offset= compress(memory);

**if**(get\_proc\_size(my\_proc)+curr\_offset > memory->size)

{

**return** false;

}

}

memory->volume\_count++;

memory->volumes = realloc(memory->volumes, memory->volume\_count\*(**sizeof**(volume)));

memory->volumes[memory->volume\_count-1].base = curr\_offset;

memory->volumes[memory->volume\_count-1].procs\_count = 0;

memory->volumes[memory->volume\_count-1].procs = malloc(memory->volumes[memory->volume\_count-1].procs\_count\***sizeof**(proc\*));

memory->volumes[memory->volume\_count-1].size = 0;

add\_proc\_to\_volume(my\_proc,&memory->volumes[memory->volume\_count-1]);

**return** true;

}

bool **add\_to\_volume**(phys\_mem\* memory, proc\* my\_proc, size\_t volume\_id)

{

**if**(volume\_id<memory->volume\_count)

{

**if**(volume\_id + 1 < memory->volume\_count){

**if**(get\_proc\_size(my\_proc)+memory->volumes[volume\_id].base+memory->volumes[volume\_id].size<= memory->volumes[volume\_id+1].base)

{

add\_proc\_to\_volume(my\_proc,&memory->volumes[volume\_id]);

}

**else**

{

compress(memory);

**if**(give\_me\_space\_after(memory,get\_proc\_size(my\_proc),volume\_id))

{

add\_proc\_to\_volume(my\_proc,&memory->volumes[volume\_id]);

}

**else**

{

**return** false;

}

}

}

**else**

{

**if**(get\_proc\_size(my\_proc)+memory->volumes[volume\_id].base+memory->volumes[volume\_id].size <= memory->size)

{

add\_proc\_to\_volume(my\_proc,&memory->volumes[volume\_id]);

}

**else**

{

size\_t offset = compress(memory);

**if**(offset + get\_proc\_size(my\_proc) <= memory->size)

{

add\_proc\_to\_volume(my\_proc,&memory->volumes[volume\_id]);

}

**else**

{

**return** false;

}

}

}

}

**else**

{

**return** false;

}

**return** true;

}

bool **give\_me\_space\_after**(phys\_mem\* memory, size\_t size, size\_t volume\_id)

{

**if**(memory->volumes[memory->volume\_count-1].base + memory->volumes[memory->volume\_count-1].size + size <= memory->size)

{

**for**(size\_t i = volume\_id+1; i < memory->volume\_count; i++)

{

memory->volumes[i].base += size;

memory->volumes[i].size += size;

}

**return** true;

}

**return** false;

}

size\_t **compress**(phys\_mem\* memory)

{

**for**(size\_t i = 1; i < memory->volume\_count; i++)

{

**if**(memory->volumes[i].base > (memory->volumes[i-1].base+memory->volumes[i-1].size))

{

size\_t diff = memory->volumes[i].base - (memory->volumes[i-1].base+memory->volumes[i-1].size);

memory->volumes[i].base -= diff;

}

}

size\_t curr\_offset = memory->volumes[memory->volume\_count-1].base + memory->volumes[memory->volume\_count-1].size;

**return** curr\_offset;

}

void **delete\_proc\_from\_volume**(phys\_mem\* memory, proc\* my\_proc, size\_t volume\_id)

{

**for**(size\_t i = 0; i < memory->volumes[volume\_id].procs\_count; i++)

{

**if**(memory->volumes[volume\_id].procs[i] == my\_proc)

{

**if**(memory->volumes[volume\_id].procs\_count == 1){

delete\_volume(memory,volume\_id);

**break**;

}

**else**

{

memory->volumes[volume\_id].procs\_count--;

**for**(size\_t k = i; k < memory->volumes[volume\_id].procs\_count; k++)

{

memory->volumes[volume\_id].procs[k] = memory->volumes[volume\_id].procs[k+1];

}

memory->volumes[volume\_id].procs = realloc(memory->volumes[volume\_id].procs,memory->volumes[volume\_id].procs\_count\***sizeof**(proc));

}

}

}

}

void **delete\_volume**(phys\_mem\* memory, size\_t volume\_id)

{

memory->volume\_count--;

**for**(size\_t i = volume\_id; i < memory->volume\_count; i++)

{

memory->volumes[i] = memory->volumes[i+1];

}

memory->volumes = realloc(memory->volumes,memory->volume\_count\***sizeof**(volume));

}

==> process.c <==

#**include** "process.h"

#**include** <math.h>

size\_t segs\_num = 0;

proc\* **generate\_procs**(size\_t count)

{

proc\* procs = malloc(count\***sizeof**(proc));

**for**(size\_t i = 0; i < count; i++)

{

#**ifndef** NO\_RANDOM

size\_t segments\_count = (size\_t)rand() % 5 + 1;

size\_t lifetime = (size\_t)rand()%10+1;

#**endif**

#**ifdef** NO\_RANDOM

size\_t segments\_count = 1;

size\_t lifetime = 1;

#**endif**

procs[i].lifetime = lifetime;

procs[i].segments\_count = segments\_count;

procs[i].segments = malloc(segments\_count\***sizeof**(seg));

**for**(size\_t k = 0; k < segments\_count; k++)

{

#**ifndef** NO\_RANDOM

size\_t idents\_count = (size\_t)rand() % 3 + 1;

#**endif**

#**ifdef** NO\_RANDOM

size\_t idents\_count = 3;

#**endif**

procs[i].segments[k].idents\_count = idents\_count;

procs[i].segments[k].idents = malloc(idents\_count\***sizeof**(ident));

size\_t size\_of\_segment = 0;

procs[i].segments[k].id = segs\_num++;

**for**(size\_t j = 0; j < idents\_count; j++)

{

char\* ident\_names[10] = {"ident0","ident1","ident2","ident3","ident4","ident5","ident6","ident7","ident8","ident9"};

#**ifndef** NO\_RANDOM

size\_t size = (size\_t)pow(2,rand()%3+1);

size\_t name\_id = (size\_t)rand()%9;

#**endif**

#**ifdef** NO\_RANDOM

size\_t size = 4;

size\_t name\_id = j;

#**endif**

procs[i].segments[k].idents[j].name = ident\_names[name\_id];

procs[i].segments[k].idents[j].size = size;

procs[i].segments[k].idents[j].addr.segment\_id = procs[i].segments[k].id;

**if**(j == 0)

procs[i].segments[k].idents[j].addr.addr\_in\_seg = 0;

**else**

procs[i].segments[k].idents[j].addr.addr\_in\_seg = procs[i].segments[k].idents[j-1].size;

size\_of\_segment += size;

}

procs[i].segments[k].size = size\_of\_segment;

}

}

**return** procs;

}

size\_t **get\_proc\_size**(proc\* my\_proc)

{

size\_t size = 0;

**for**(size\_t i = 0; i < my\_proc->segments\_count; i++)

size += my\_proc->segments[i].size;

**return** size;

}

==> segment.c <==

#**include** "segment.h"

#**include** <string.h>

#**include** <stdlib.h>

virt\_addr\* get\_virt\_addr(char\* name, seg my\_seg)

{

**for**(size\_t i = 0; i < my\_seg.idents\_count; i++)

{

**if**(strcmp(my\_seg.idents->name, name) == 0)

**return** &my\_seg.idents->addr;

}

exit(EXIT\_FAILURE);

}

==> virtual\_address.c <==

#**include** <stdio.h>

#**include** "virtual\_address.h"

void print\_virt\_addr(virt\_addr\* va)

{

printf("%2llu..%4llu",va->segment\_id,va->addr\_in\_seg);

}

==> volume.c <==

#**include** "volume.h"

#**include** <stdio.h>

#**include** <string.h>

void add\_proc\_to\_volume(proc\* my\_proc, volume\* my\_volume)

{

my\_volume->procs\_count++;

my\_volume->procs = realloc(my\_volume->procs,my\_volume->procs\_count\***sizeof**(proc\*));

my\_volume->procs[my\_volume->procs\_count-1] = my\_proc;

my\_volume->size += get\_proc\_size(my\_proc);

}

void **delete\_proc\_from\_volume\_by\_id**(size\_t id\_proc, volume\* my\_volume)

{

my\_volume->procs\_count--;

my\_volume->size -= get\_proc\_size(my\_volume->procs[id\_proc]);

**for**(size\_t i = id\_proc; i < my\_volume->procs\_count;i++)

{

my\_volume->procs[i] = my\_volume->procs[i+1];

}

my\_volume->procs = realloc(my\_volume->procs,my\_volume->procs\_count\*(**sizeof**(proc)));

}

size\_t **whoami\_physically**(proc\* my\_proc, volume\* my\_volume)

{

**for**(size\_t i = 0; i < my\_volume->procs\_count; i++)

{

**if**(my\_volume->procs[i] == my\_proc)

**return** i;

}

printf("Tried to access to proc from another volume\n");

exit(EXIT\_FAILURE);

}

phys\_addr **get\_phys\_addr**(proc\* my\_proc, volume\* my\_volume)

{

size\_t id = whoami\_physically(my\_proc,my\_volume);

size\_t offset = 0;

**for**(size\_t i = 0; i < id; i++)

{

offset+=get\_proc\_size(my\_volume->procs[i]);

}

phys\_addr phys = {.base = my\_volume->base,

.offset = offset};

**return** phys;

}

phys\_addr **get\_phys\_addr\_ident**(char\* name, seg\* segment, proc\* my\_proc,volume\* my\_volume)

{

phys\_addr procs\_one = get\_phys\_addr(my\_proc,my\_volume);

procs\_one.offset = 0;

size\_t offset = 0;

**for**(size\_t i = 0; i < my\_proc->segments\_count; i++)

{

offset += my\_proc->segments[i].size;

**if**(my\_proc->segments[i].id == segment->id)

{

**for**(size\_t k = 0; k < my\_proc->segments[i].idents\_count; k++)

{

**if**(strcmp(my\_proc->segments[i].idents[k].name,name) == 0)

{

procs\_one.offset+= offset;

**return** procs\_one;

}

offset += my\_proc->segments[i].idents[k].size;

}

}

}

printf("ERROR: cannot find specified name or segment or proc or volume or etc\n");

exit(EXIT\_FAILURE);

}

==> config.h <==

#**ifndef** CONFIG\_H

#**define** CONFIG\_H

//#define NO\_RANDOM

#**define** RAM\_SIZE 128

#**endif**

==> identifier.h <==

#**include** <stdlib.h>

#**include** "virtual\_address.h"

#**ifndef** IDENTIFIER\_H

#**define** IDENTIFIER\_H

**struct** Identifiers{

char\* name;

size\_t size; // 2, 4, 8, 2^n

virt\_addr addr;

};

**typedef** **struct** **Identifiers** **ident**;

#**endif**

==> memory\_controller.h <==

#**include** "process.h"

#**ifndef** MEMORY\_CONTROLLER\_H

#**define** MEMORY\_CONTROLLER\_H

**struct** MemoryController

{

size\_t max\_size;

size\_t used\_memory;

size\_t free\_memory;

size\_t procs\_count;

proc\* procs;

};

**typedef** **struct** **MemoryController** **mem\_control**;

#**endif**

==> physical\_address.h <==

#**include** <stdlib.h>

#**ifndef** PHYSICAL\_ADDRESS\_H

#**define** PHYSICAL\_ADDRESS\_H

**struct** PhysicalAddress

{

size\_t base;

size\_t offset;

};

**typedef** **struct** **PhysicalAddress** **phys\_addr**;

void **print\_phys\_addr**(phys\_addr\* pa);

#**endif**

==> physical\_memory.h <==

#**include** "volume.h"

#**include** <stdbool.h>

#**ifndef** PHYSICAL\_MEMORY

#**define** PHYSICAL\_MEMORY

**struct** PhysicalMemory

{

size\_t size;

size\_t volume\_count;

volume\* volumes;

};

**typedef** **struct** **PhysicalMemory** **phys\_mem**;

size\_t **compress**(phys\_mem\* memory);

bool **give\_me\_space\_after**(phys\_mem\* memory, size\_t size, size\_t volume\_id);

bool **add\_to\_volume**(phys\_mem\* memory, proc\* my\_proc, size\_t volume\_id);

bool **create\_volume**(phys\_mem\* memory, proc\* my\_proc);

void **delete\_proc\_from\_volume**(phys\_mem\* memory, proc\* my\_proc, size\_t volume\_id);

void **delete\_volume**(phys\_mem\* memory, size\_t volume\_id);

#**endif**

==> process.h <==

#**include** "segment.h"

#**ifndef** PROCESS\_H

#**define** PROCESS\_H

**struct** Process{

seg\* segments; // Not NULL

size\_t segments\_count; // >0

size\_t lifetime; // >0

};

**typedef** **struct** **Process** **proc**;

proc\* **generate\_procs**(size\_t count);

size\_t **get\_proc\_size**(proc\* my\_proc);

#**endif**

==> segment.h <==

#**include** "identifier.h"

#**ifndef** SEGMENT\_H

#**define** SEGMENT\_H

**struct** Segment{

size\_t id; // >= 0

ident\* idents; // Not NULL

size\_t idents\_count; // >0

size\_t size; // >0

};

**typedef** **struct** **Segment** **seg**;

virt\_addr\* **get\_virt\_addr**(char\* name, seg my\_seg);

#**endif**

==> virtual\_address.h <==

#**include** <stdlib.h>

#**ifndef** VIRTUAL\_ADDRESS\_H

#**define** VIRTUAL\_ADDRESS\_H

**struct** VirtualAddress

{

size\_t segment\_id; // >= 0

size\_t addr\_in\_seg; // >= 0

};

**typedef** **struct** **VirtualAddress** **virt\_addr**;

void **print\_virt\_addr**(virt\_addr\* va);

#**endif**

==> volume.h <==

#**include** "process.h"

#**include** "physical\_address.h"

#**ifndef** VOLUME\_H

#**define** VOLUME\_H

**struct** Volume{

size\_t base;

size\_t size;

proc\*\* procs;

size\_t procs\_count;

};

**typedef** **struct** **Volume** **volume**;

phys\_addr **get\_phys\_addr**(proc\* proc, volume\* volume);

phys\_addr **get\_phys\_addr\_ident**(char\* name, seg\* segment, proc\* proc,volume\* volume);

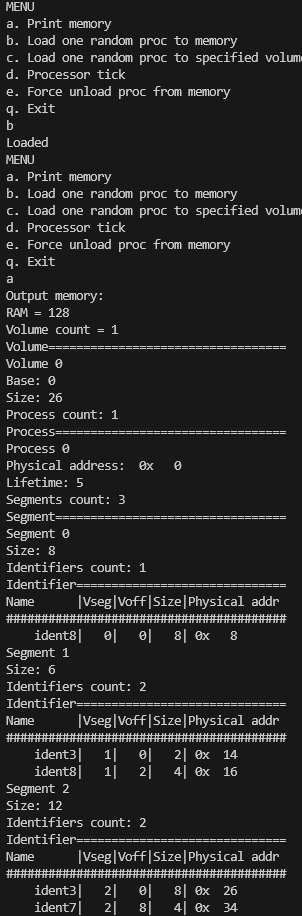
size\_t **whoami\_physically**(proc\* proc, volume\* volume);

void **delete\_proc\_from\_volume\_by\_id**(size\_t id\_proc, volume\* volume);

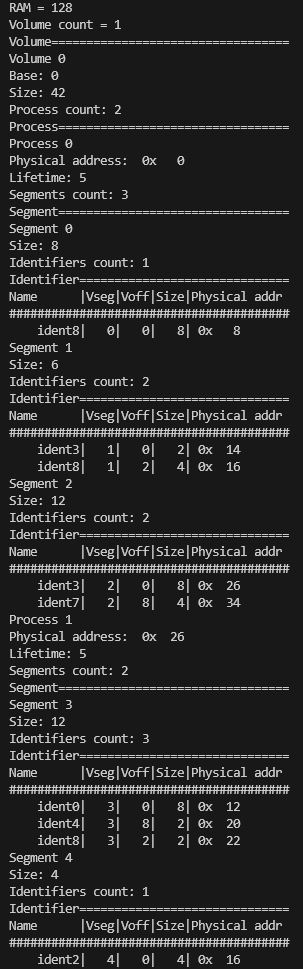
void **add\_proc\_to\_volume**(proc\* proc, volume\* volume);

#**endif**

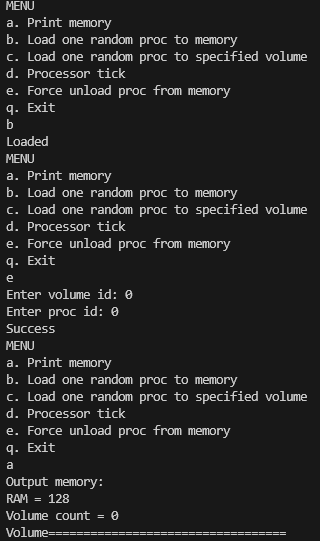
*Скріншоти програми:*

**

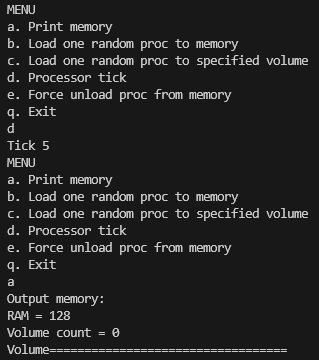
*Додавання процессу(зі створенням розділу)*

**

*Додавання процесу(в існуючий розділ)*

**

*Видалення процесу з розділу(якщо був один процес, то і розділу)*

**

*Видалення процесів з пам’яті після кінця їх роботи*

*Висновок:*

Під час виконання лабораторної роботи ознайомились з існуючими способами структурування пам’яті, алгоритмами керування пам’яттю та перетворення віртуальних адрес у фізичні адреси. Була написана програма, що моделює процес управління пам’яттю, а саме розподілу пам’яті для процесів, перетворення віртуальних адрес у фізичні, пошук пам’яті за запитами процесів, вивільнення пам’яті. Була продемонстрована робота моделі з використанням основних операцій з пам’яттю. Завдання операцій реалізовано за допомогою меню.

За варіантом були змодельовані переміщувальні розділи (без використання зовнішньої пам’яті). Кількість розділів була менше або дорівнює кількості процесів. Якщо розділ було неможливо розмістити у пам’яті(чи процес додати до розділу), то виконувалася процедура «стискування» в напрямку молодших адрес. Процеси мали загальну чергу до розділів пам’яті. Був використаний сегментований адресний простір. Розміри процесів, кількість сегментів, кількість ідентифікаторів та їх розміри задавалися випадково.

Істотним недоліком переміщувальних розділів є фрагментація пам’яті, з метою зменшення якого виконується процедура «стискування», але в результаті необхідна реалізація динамічного перетворення адрес, що вимагає значних накладних витрат при значній кількості процесів. При моделюванні роботу з перетворення віртуальної адреси у фізичну виконує динамічний завантажувальник. Віртуальний адресний простір складається з пари чисел, де перше сегмент(Vseg, Virtual Segment), а друге зміщення всередині сегмента(Voff, Virtual Offset).

При розробці програми, що моделює управління пам’яттю була реалізована функціональність звернення до ідентифікатора з перетворенням адреси спочатку на віртуальну, а потім і на фізичну. З урахуванням завдання за варіантом, де нема використання зовнішньої пам’яті, при нестачі пам’яті не відбувається витискання даних на диск, процес не запускається, видається помилка нестачі пам’яті. Кількість зайнятої пам’яті та вільної завжди відслідковується. Адреси програми налаштовані на конкретну область фізичної пам’яті, розмір якої можна задати в програмному коді.