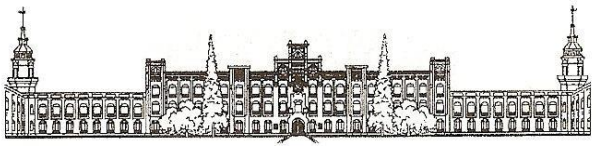

Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут»

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. Написати програму, що моделює процес управління пам'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.

2. 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 – наглядна інформація про поточний розподіл пам'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","ident
7","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:

```
MENU
a. Print memory
b. Load one random proc to memory
c. Load one random proc to specified volum
d. Processor tick
e. Force unload proc from memory
q. Exit
b
Loaded
MENU
a. Print memory
b. Load one random proc to memory
c. Load one random proc to specified volum
d. Processor tick
e. Force unload proc from memory
q. Exit
a
Output memory:
RAM = 128
Volume count = 1
Volume==================================
Volume 0
Base: 0
Size: 26
Process count: 1
Process==================================
Process 0
Physical address:  0x   0
Lifetime: 5
Segments count: 3
Segment==================================
Segment 0
Size: 8
Identifiers count: 1
Identifier===============================
Name      |Vseg|Voff|Size|Physical addr
########################################
    ident8|   0|   0|   8| 0x   8
Segment 1
Size: 6
Identifiers count: 2
Identifier===============================
Name      |Vseg|Voff|Size|Physical addr
########################################
    ident3|   1|   0|   2| 0x  14
    ident8|   1|   2|   4| 0x  16
Segment 2
Size: 12
Identifiers count: 2
Identifier===============================
Name      |Vseg|Voff|Size|Physical addr
########################################
    ident3|   2|   0|   8| 0x  26
    ident7|   2|   8|   4| 0x  34
```

Додавання процессу(зі створенням розділу)

```
RAM = 128
Volume count = 1
Volume==================================
Volume 0
Base: 0
Size: 42
Process count: 2
Process=================================
Process 0
Physical address:  0x   0
Lifetime: 5
Segments count: 3
Segment=================================
Segment 0
Size: 8
Identifiers count: 1
Identifier==============================
Name      |Vseg|Voff|Size|Physical addr
########################################
    ident8|   0|   0|   8| 0x   8
Segment 1
Size: 6
Identifiers count: 2
Identifier==============================
Name      |Vseg|Voff|Size|Physical addr
########################################
    ident3|   1|   0|   2| 0x  14
    ident8|   1|   2|   4| 0x  16
Segment 2
Size: 12
Identifiers count: 2
Identifier==============================
Name      |Vseg|Voff|Size|Physical addr
########################################
    ident3|   2|   0|   8| 0x  26
    ident7|   2|   8|   4| 0x  34
Process 1
Physical address:  0x  26
Lifetime: 5
Segments count: 2
Segment=================================
Segment 3
Size: 12
Identifiers count: 3
Identifier==============================
Name      |Vseg|Voff|Size|Physical addr
########################################
    ident0|   3|   0|   8| 0x  12
    ident4|   3|   8|   2| 0x  20
    ident8|   3|   2|   2| 0x  22
Segment 4
Size: 4
Identifiers count: 1
Identifier==============================
Name      |Vseg|Voff|Size|Physical addr
########################################
    ident2|   4|   0|   4| 0x  16
```

Додавання процесу(в існуючий розділ)

```
MENU
a. Print memory
b. Load one random proc to memory
c. Load one random proc to specified volume
d. Processor tick
e. Force unload proc from memory
q. Exit
b
Loaded
MENU
a. Print memory
b. Load one random proc to memory
c. Load one random proc to specified volume
d. Processor tick
e. Force unload proc from memory
q. Exit
e
Enter volume id: 0
Enter proc id: 0
Success
MENU
a. Print memory
b. Load one random proc to memory
c. Load one random proc to specified volume
d. Processor tick
e. Force unload proc from memory
q. Exit
a
Output memory:
RAM = 128
Volume count = 0
Volume================================
```

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

```
MENU
a. Print memory
b. Load one random proc to memory
c. Load one random proc to specified volume
d. Processor tick
e. Force unload proc from memory
q. Exit
d
Tick 5
MENU
a. Print memory
b. Load one random proc to memory
c. Load one random proc to specified volume
d. Processor tick
e. Force unload proc from memory
q. Exit
a
Output memory:
RAM = 128
Volume count = 0
Volume================================
```

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.