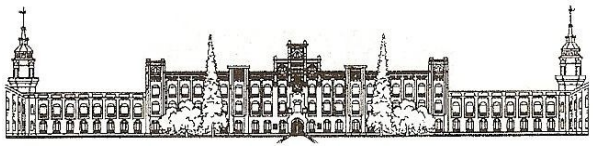

Міністерство освіти і науки України Національний технічний університет України

«Київський політехнічний інститут»

Лабораторна робота №3

з дисципліни «Введення до операційних систем»

Виконав студент групи: КВ-22

ПІБ: Крутогуз Максим Ігорович

Перевірив:

_____

Київ 2025

Варіант завдання:

№пп	ПІБ	Варіанти завдань ЛАБ
1	Бондарева Валерія	9
2	Вакульчук Ярослав	10
3	Вороняк Максим	11
4	Гарматюк Катерина	12
5	Гончар Вікторія	12
6	Гречишкіна Катерина	13
7	Деркач Андрій	14
8	Землянський Едуард	15
9	Кобан Ілля	1
10	Ковкін Владислав	2
11	Кошулько Владислав	3
12	Крутогуз Максим	4
13	Лисенко Віталій	5
14	Лукащук Юлія	6
15	Марчук Дмитро	7
16	Міндер Вадим	8
17	Некрасова Поліна	9
18	Пляченко Олександр	10
19	Приходько Станіслав	11
20	Редько Катерина	12
21	Романченко Вікторія	13
22	Савельєв Олександр	14

4-ий варіант:

Варіанти завдання, що відповідають різним способам організації пам'яті, представлені у табл. 3.1.

При моделировании алгоритмов без использования внешней памяти считать что исходные адреса каждого незагруженного процесса начинается с 0000..00, размеры каждого из процессов должны задаваться произвольно. Каждому адресу незагруженного процесса по запросу следует указать реальный адрес памяти если процесс загружен.

Загрузку и выгрузку (при необходимости) процессов выполнять в соответствии с

конкретно заданной очередью.

При моделировании алгоритмов с использованием внешней памяти следует задать виртуальное адресное пространство каждого процесса. Обращение к памяти выполнять по виртуальному адресу, осуществлять при необходимости загрузку и выгрузку соответствующих частей процессов, распечатывая значения конкретного физического адреса, который соответствует заданному виртуальному.

При моделировании алгоритмов работы КЭШ-памяти адресное пространство основной и КЭШ-памяти может задаваться произвольно при соотношении объемов КЭШ-памяти и основной не менее 1:10. Поиск, запись и замещение информации в КЭШ-памяти должно выполняться путем задания искомых адресов основной памяти.

4.	**Переміщувальні розділи** (без використання зовнішньої пам'яті). Кількість розділів - меньша, ніж кількість процесів. Якщо черговий розділ неможливо розмістити у пам'яті, виконується процедура «стискання» в напрямку **молодших адрес.** Процеси утворюють загальну чергу до розділів пам'яті. Використовується **лінійний** адресний простір. Розміри процесів задаються випадково.

Корисні посилання:

- Виконуваний файл
- Github

```
Enter your next command: h
Information about available commands:
h           Shows information about available commands
l <index>   Attempt for loading an unloaded process
u <index>   Attempt for unloading an loaded process
m           Shows information about memory allocation
p           Shows information about status of all processes
e           Exits
```

Команди програми

Приклад тестування алгоритму

```
Enter your next command: p
PID  isLoaded  Memory address Size
1    true      0              290
3    true      290            30
7    true      880            60
9    true      940            200
14   true      1140           1000
13   true      2140           720
12   true      2860           690
5    false     -1             560
2    false     -1             990
4    false     -1             150
6    false     -1             170
8    false     -1             750
10   false     -1             820
11   false     -1             830
15   false     -1             780
Enter your next command: m
PID  Memory address
1    0:290
3    290:320
7    880:940
9    940:1140
14   1140:2140
13   2140:2860
12   2860:3550
Enter your next command: l 15
[-1]
Load process executed successfully
```

```
Enter your next command: p
PID  isLoaded  Memory address Size
1    true      0              290
3    true      290            30
7    true      320            60
9    true      380            200
14   true      580            1000
13   true      1580           720
12   true      2300           690
15   true      2990           780
5    false     -1             560
2    false     -1             990
4    false     -1             150
6    false     -1             170
8    false     -1             750
10   false     -1             820
11   false     -1             830
Enter your next command: m
PID  Memory address
1    0:290
3    290:320
7    320:380
9    380:580
14   580:1580
13   1580:2300
12   2300:2990
15   2990:3770
```

Додавання процесу і під час цього процесу відбувається переміщення всіх розділів для зменшення фрагментація та ймовірно можливість додати новий процес навіть якщо під час фрагментації цього зробити не можна було.

Код

```cpp
// App.cpp
#include <iostream>

#include "Controller.hpp"

using namespace std;

int main() {
    Controller controler;

    controler.run();
```

```
    return 0;
}


// Controller.cpp
#include "Controller.hpp"

Controller::Controller() : _model(Model()) {
    _model.init(15, 4000);
}

void Controller::run() {
    while (true)
    {
        cout << "Enter your next command: ";
        string commandString;
        getline(cin, commandString);

        istringstream ss(commandString);

        vector<string> command;
        string token;

        while (ss >> token) {
            command.push_back(token);
        }

        try {
            if (command.size() <= 0 || command.size() > 2) {
                throw "prohibited";
            }

            string commandName = command[0];
            if (command.size() == 1) {
                if (commandName == "h") {
                    help();
                } else if (commandName == "m") {
                    memoryStatus();
                } else if (commandName == "p") {
                    processStatus();
                } else if (commandName == "e") {
                    exit(0);
                } else {
                    throw "prohibited";
                }
            } else {
                if (commandName == "l") {
                    loadProcess(stoi(command[1]));
                } else if (commandName == "u") {
                    unloadProcess(stoi(command[1]));
                } else {
                    throw "prohibited";
                }
            }
        } catch(...) {
            cout << "Incorect command, for getting info, please type h" << endl;
        }

        // cout << "You: " << commandString << endl;
    }
```

```
}


void Controller::help() {
    int indent = 12;
    cout << "Information about available commands:" << endl;
    cout << left << setw(indent) << "h" << "Shows information about available
commands" << endl;
    cout << left << setw(indent) << "l <index>" << "Attempt for loading an
unloaded process" << endl;
    cout << left << setw(indent) << "u <index>" << "Attempt for unloading an
loaded process" << endl;
    cout << left << setw(indent) << "m" << "Shows information about memory
allocation" << endl;
    cout << left << setw(indent) << "p" << "Shows information about status of
all processes" << endl;
    cout << left << setw(indent) << "e" << "Exits" << endl;
}


void Controller::memoryStatus() {
    vector<Process> *processes = _model.getProcesses();

    bool isAnyProcessLoaded = false;

    for (auto process : *processes) {
        if (process.isLoaded) {
            isAnyProcessLoaded = true;
        }
    }

    if (isAnyProcessLoaded) {
        cout << setw(5) << left << "PID" << setw(30) << left << "Memory address"
<< endl;
    } else {
        cout << "There are no process running now" << endl;
    }

    for (auto process : *processes) {
        if (process.isLoaded) {
            ostringstream ss;
            ss << process.memoryAddress << ':' << process.memoryAddress +
process.size;

            cout << setw(5) << left << process.id << setw(30) << left <<
ss.str() << endl;
        }
    }
}


void Controller::processStatus() {
    vector<Process> *processes = _model.getProcesses();
    cout << setw(5) << "PID" << setw(10) << "isLoaded"<< setw(15) << "Memory
address"  << setw(10) << "Size" << endl;

    for (auto process : *processes) {
        cout << setw(5) << process.id << setw(10) << boolalpha <<
process.isLoaded << setw(15) << process.memoryAddress << setw(10) <<
process.size << endl;
    }
```

```
}


void Controller::loadProcess(int index) {
    bool isLoaded = _model.load(index);

    if (isLoaded) {
        cout << "Load process executed successfully" << endl;
    } else {
        cout << "Load process failed" << endl;
    }
}


void Controller::unloadProcess(int index) {
    bool isLoaded = _model.unload(index);

    if (isLoaded) {
        cout << "Unload process executed successfully" << endl;
    } else {
        cout << "Unload process failed" << endl;
    }
}


// Controller.hpp
#ifndef CONTROLLER_HPP
#define CONTROLLER_HPP

#include <iostream>
#include <string>
#include <sstream>
#include <vector>
#include <iomanip>
#include <cstdlib>

#include "Model.hpp"

using namespace std;

class Controller {
    public:
    Controller();
    void run();

    private:
    void help();
    void memoryStatus();
    void processStatus();
    void loadProcess(int index);
    void unloadProcess(int index);

    Model _model;
};

#endif


// Global.cpp
#include "Global.hpp"
```

```cpp
unsigned seed = chrono::system_clock::now().time_since_epoch().count();

default_random_engine generator(seed);


// Global.hpp
#ifndef GLOBAL_HPP
#define GLOBAL_HPP

#include <random>
#include <chrono>

using namespace std;

extern default_random_engine generator;

#endif


// Model.cpp
#include "Model.hpp"

Model::Model () : _memorySize(0), _insertionMemory(0) {}


void Model::init(int processCount, int memorySize) {
    uniform_int_distribution<int> distribution(1, 100);

    for (int i = 0; i < processCount; i++) {
        _processes.push_back({
            i + 1,
            false,
            -1,
            distribution(generator) * 10
        });
    }

    _memorySize = memorySize;
}

vector<Process>* Model::getProcesses() {
    return &_processes;
}

bool Model::load(int index) {
    int processIndex = findProcessById(index);

    if (processIndex == -1) {
        return false;
    }

    auto proc = _processes[processIndex];

    if (proc.isLoaded) {
        return false;
    }


    int processSize = proc.size;

    int insertIndex = findIndexForLoad(processSize);
```

```
    // cout << '[' << insertIndex << ']' << endl;

    if (insertIndex == -1) {
        compression();
    }

    insertIndex = findIndexForLoad(processSize);

    if (insertIndex == -1) {
        return false;
    }

    Process processBuf =  _processes[processIndex];
    processBuf.isLoaded = true;
    processBuf.memoryAddress = _insertionMemory;

    for (int i = processIndex; i > insertIndex; --i) {
        _processes[i] = _processes[i - 1];
    }

    _processes[insertIndex] = processBuf;

    return true;
}

bool Model::unload(int index) {
    int processIndex = findProcessById(index);

    if (processIndex == -1) {
        return false;
    }

    auto proc = _processes[processIndex];

    if (!proc.isLoaded) {
        return false;
    }

    int insertIndex = _processes.size() - 1;
    for (int i = 0; i < (int)_processes.size(); i++) {
        if (!_processes[i].isLoaded) {
            insertIndex = i;
            break;
        }
    }
    // cout << '[' << insertIndex << ']' << endl;

    Process processBuf = _processes[processIndex];
    processBuf.isLoaded = false;
    processBuf.memoryAddress = -1;

    if (processIndex < insertIndex) {
        for (int i = processIndex; i < insertIndex; i++) {
            _processes[i] = _processes[i + 1];
        }
        _processes[insertIndex - 1] = processBuf;
    } else {
        _processes[processIndex] = processBuf;
    }
```

```
    return true;
}


//---------private----------

int Model::findProcessById(int pid) {
    int index = 0;
    for (auto process : _processes) {
        if (process.id == pid) {
            return index;
        }
        index++;
    }
    return -1;
}


int Model::findIndexForLoad(int processSize) {
    int startAddress = 0;
    _insertionMemory = 0;

    int loadedProcesses = 0;
    for (size_t i = 0; i < _processes.size(); ++i) {
        if (_processes[i].isLoaded) {
            if (startAddress + processSize <= _processes[i].memoryAddress) {
                return i;
            } else {
                startAddress = _processes[i].memoryAddress + _processes[i].size;
                _insertionMemory = startAddress;
            }
            loadedProcesses++;
        }
    }

    if (startAddress + processSize <= _memorySize) {
        return loadedProcesses;
    }

    return -1;
}


void Model::compression() {
    int currentAddress = 0;

    for (auto &proc : _processes) {
        if (proc.isLoaded) {
            proc.memoryAddress = currentAddress;
            currentAddress += proc.size;
        }
    }
}


// Model.hpp
#ifndef MODEL_HPP
#define MODEL_HPP

#include <vector>
#include <random>
```

```cpp
#include <iostream>

#include "Global.hpp"

using namespace std;

typedef struct {
    int id;
    bool isLoaded;
    int memoryAddress;
    int size;
} Process;

class Model {
    public:
    Model();
    void init(int processCount, int memorySize);
    vector<Process>* getProcesses();
    bool load(int index);
    bool unload(int index);

    private:
    int findProcessById(int pid);
    int findIndexForLoad(int processSize);
    void compression();

    int _memorySize;
    int _insertionMemory;
    vector<Process> _processes;
};

#endif
```