МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение высшего образования
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

КАФЕДРА КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ И ПРОГРАММНОЙ ИНЖЕНЕРИИ

РУКОВОДИТЕЛЬ

старший преподаватель		М. Д. Поляк
должность, уч. степень, звание	подпись, дат	инициалы, фамилия

ЛАБОРАТОРНАЯ РАБОТА

«№4. Управление памятью»

по дисциплине:

Операционные системы

РАБОТУ ВЫПОЛНИЛ
СТУДЕНТ ГР. Z1431 _____ А.С. Зайцев
подпись, дата | инициалы, фамилия

Санкт-Петербург 2025

Цель работы

Знакомство с принципами организации виртуальной памяти.

Задание на лабораторную работу

В данной работе необходимо реализовать фрагмент диспетчера памяти и часть функционала операционной системы, отвечающего за замещение страниц при возникновении ошибок отсутствия страниц. Для упрощения работы предполагается использование линейной инвертированной таблицы страниц, работу с которой необходимо реализовать в виде программы. Также для простоты предполагается, что в системе имеется один единственный процесс, поэтому идентификатор процесса в инвертированной таблице страниц не хранится. Входные данные представляют собой последовательность операций обращения к памяти, выходные данные - состояние инвертированной таблицы страниц после каждой операции обращения к памяти.

1. Вычислить номер варианта по списку в журнале и сохранить его в файл TASKID.txt в репозитории.
2. Написать программу на языке C++ в соответствии со следующей спецификацией.
    a. Входные данные:
        i. Аргумент командной строки (число): номер алгоритма замещения страниц, который должна использовать программа. Принимает значения 1 или 2, соответствующие двум алгоритмам замещения страниц, заданным по варианту.
        ii. Перечень инструкций обращения к памяти, считываемый программой из стандартного потока ввода. На каждой строке не более одной инструкции. Инструкция состоит из двух чисел, разделенных пробелом, например: 0 1. Первое число обозначает тип операции доступа к памяти: 0 - чтение и 1 - запись. Второе число является номером виртуальной страницы, к которой происходит обращение.
    b. Выходные данные:
        i. Для каждой операции обращения к памяти, информация о которой поступила на вход программы, на выходе должна быть сгенерирована строка, содержащая содержимое инвертированной таблицы страниц в виде последовательности номеров виртуальных страниц, разделенных пробелом. Если какая-либо из записей в таблице страниц отсутствует (таблица страниц не заполнена до конца), вместо номера виртуальной страницы необходимо вывести символ #.
3. Весь код поместить в файле lab4.cpp. Код должен корректно компилироваться командой g++ lab4.cpp -o lab4 -std=c++11. Настоятельно рекомендуется использовать стандартную библиотеку STL. Полезными могут быть контейнеры list, vector, bitset и др.

4. Если в работе алгоритма замещения страниц используется бит R, то необходимо реализовать эмуляцию прерывания таймера. Для этого через каждые 5 операций обращения к памяти необходимо запускать обработчик данного прерывания. Значения битов R по прерыванию таймера сбрасываются.
5. Для алгоритмов, использующих счетчик (NFU, Aging): если несколько страниц имеют одинаковое значение счетчика, одна из них выбирается случайным образом. При повторной загрузке страницы в память ее счетчик обнуляется. В алгоритме старения счетчик имеет размер 1 байт. В алгоритме NFU счетчик имеет размер не меньше 4 байт.
6. Во всех алгоритмах, использующих датчик случайных чисел (Random, NRU, NFU, Aging, ...), разрешается использовать **только** функцию int uniform_rnd(int a, int b), объявленную в файле lab4.h. Данная функция генерирует случайное целое число с равномерным распределением из диапазона [a, b]. Использование других функций для работы со случайными числами запрещено!
7. В качестве системного времени в алгоритме рабочего набора следует использовать количество инструкций доступа к памяти, обработанных с момента запуска программы.
8. После успешного прохождения локальных тестов необходимо загрузить код в репозиторий на гитхабе.
9. Сделать выводы об эффективности реализованных алгоритмов замещения страниц. Сравнить количество ошибок отсутствия страниц, генерируемых на тестовых данных при использовании каждого алгоритма.
10. Подготовить отчет о выполнении лабораторной работы и загрузить его под именем report.pdf в репозиторий. В случае использования системы компьютерной верстки LaTeX также загрузить исходный файл report.tex.

Вариант 5

5	6	FIFO	NFU

6 - страниц

Алгоритмы: FIFO и NFU

Результат выполнения работы

```
gently@zaytsev-a-mb os-task4-gently-whitesnow % bash run_test.sh
Found shunit2-2.1.8
test_TASKID
TASKID is 5
test_build
test_algorithm1
test_algorithm2
test/in60.txt

Ran 4 tests.

OK
```

Подсчет отсутствия страниц

FIFO Page Faults: 52

NFU Page Faults: 54

Исходный код программы

```
#include "lab4.h"
#include <iostream>
#include <queue>
#include <vector>

int NUM_PAGES = 6;
int INTERAPTION_CALLS = 5;

struct Page {
int vpn; // Виртуальный номер страницы
bool r; // Счетчик обращений
int counter; // Счетчик для NFU
Page(int vpn) : vpn(vpn), r(true), counter(0) {}
};

void printTable(const std::vector<Page *> &table) {
for (size_t i = 0; i < table.size(); ++i) {
if (table[i]) {
std::cout << table[i]->vpn;
} else {
std::cout << "#";
}
if (i < table.size() - 1) {
std::cout << " "; // Пробел только между элементами
}
}
std::cout << std::endl;
}

void cleanupTable(std::vector<Page *> &table) {
for (auto &page : table) {
delete page;
page = nullptr;
```

```
}
}

bool tryReadFifoPage(const std::vector<Page *> &table, int vpn) {
for (auto &page : table) {
if (page && page->vpn == vpn) {
return true;
}
}
return false;
}

void replaceOrAddFifoPage(std::vector<Page *> &table,
std::queue<int> &fifoQueue, int vpn) {
if (fifoQueue.size() == NUM_PAGES) {
int toReplace = fifoQueue.front();
fifoQueue.pop();

for (auto &page : table) {
if (page && page->vpn == toReplace) {
page->vpn = vpn;
fifoQueue.push(vpn);
return;
}
}
} else {
for (auto &page : table) {
if (!page) {
page = new Page(vpn);
fifoQueue.push(vpn);
return;
}
}
}
}

void fifoAlgorithm(const std::vector<std::pair<int, int>> &operations) {
std::vector<Page *> table(NUM_PAGES, nullptr);
std::queue<int> fifoQueue;
int pageFaults = 0;

for (const auto &op : operations) {
int vpn = op.second;

if (!tryReadFifoPage(table, vpn)) {
++pageFaults;
replaceOrAddFifoPage(table, fifoQueue, vpn);
}

printTable(table);
}

cleanupTable(table);
// std::cout << "FIFO Page Faults: " << pageFaults << std::endl;
}
```

```
void interaption(std::vector<Page *> &table) {
for (auto &page : table) {
if (page && page->r) {
page->counter++;
page->r = false;
}
}
}

bool tryReadNfuPage(const std::vector<Page *> &table, int vpn) {
for (auto &page : table) {
if (page && page->vpn == vpn) {
page->r = true;
return true;
}
}
return false;
}

void handleInteraptionIfNeeded(int accessCounter, std::vector<Page *> &table) {
if (accessCounter % INTERAPTION_CALLS == 0) {
interaption(table);
}
}

Page *selectPageToReplace(const std::vector<Page *> &table, int vpn) {
unsigned int min = UINT32_MAX;
for (auto &page : table) {
if (page && page->counter < min) {
min = page->counter;
}
}

std::vector<Page *> candidates;
for (auto &page : table) {
if (page && page->counter == min) {
candidates.push_back(page);
}
}

if (candidates.size() == 1) {
return candidates[0];
} else {
int rndPos = uniform_rnd(0, (int)candidates.size() - 1);
return candidates[rndPos];
}
}

void replaceOrAddNfuPage(std::vector<Page *> &table, int vpn) {
if (std::all_of(table.begin(), table.end(), [](Page *p) { return p; })) {
Page *toReplace = selectPageToReplace(table, vpn);
toReplace->vpn = vpn;
toReplace->r = true;
toReplace->counter = 0;
} else {
```

```
for (auto &page : table) {
if (!page) {
page = new Page(vpn);
break;
}
}
}
}

void nfuAlgorithm(const std::vector<std::pair<int, int>> &operations) {
std::vector<Page *> table(NUM_PAGES, nullptr);
int pageFaults = 0;
int accessCounter = 0;

for (const auto &op : operations) {
int vpn = op.second;

accessCounter++;

if (!tryReadNfuPage(table, vpn)) {
pageFaults++;
replaceOrAddNfuPage(table, vpn);
}
handleInteraptionIfNeeded(accessCounter, table);
printTable(table);
}

cleanupTable(table);
// std::cout << "NFU Page Faults: " << pageFaults << std::endl;
}

int main(int argc, char *argv[]) {
if (argc < 2) {
std::cerr << "Usage: " << argv[0] << " <algorithm_number>\n";
return 1;
}

int algorithm = std::stoi(argv[1]);

std::vector<std::pair<int, int>> operations;
int type, vpn;
while (std::cin >> type >> vpn) {
operations.emplace_back(type, vpn);
}

// в зависимости от алгоритма вызываем соответствующую функцию
if (algorithm == 1) {
fifoAlgorithm(operations);
} else if (algorithm == 2) {
nfuAlgorithm(operations);
} else {
std::cerr << "Invalid algorithm number.\n";
return 1;
}
```

```
return 0;
}
```

Выводы

В ходе лабораторной работы были изучены принципы управления виртуальной памятью и проведена оценка эффективность различных подходов к замещению страниц. На основе моих результатов можно сделать вывод:

1. Алгоритм FIFO немного эффективнее NFU в данном случае, так как он генерирует меньше ошибок отсутствия страниц.

2. NFU может быть менее предсказуемым из-за своей зависимости от счетчиков использования, особенно при неравномерных обращениях к страницам.

3. Выбор алгоритма замещения страниц должен учитывать характер доступа к памяти и требования к производительности, так как разница в ошибках несущественна.