Державний університет “Одеська політехніка”
Інститут комп'ютерних систем
Кафедра «Комп'ютеризовані системи управління»

КУРСОВА РОБОТА

з дисципліни «Сучасні технології програмування»
Варіант 11

Студента 3 курсу, групи АТ–193
спеціальності «Автоматизація та
комп'ютерно-інтегровані технології»
Прізвище: Якубчинський К.В.
Керівник: доц. Сперанський В.О.

Національна шкала:_____
Кількість балів:_____
Оцінка: ECTS_____

Члени комісії _____ _____
(підпис) (прізвище та ініціали)
_____ _____
(підпис) (прізвище та ініціали)
_____ _____
(підпис) (прізвище та ініціали)

м. Одеса – 2021 рік

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Операционная система (ОС) – это организованная совокупность программ и данных, которая выполняет функции посредника между пользователями и компьютером. ОС служит двум целям: во-первых, сделать компьютерную систему удобной для использования, и, во-вторых, эффективно использовать аппаратные средства компьютера. ОС является *управляющей программой*. Управляющая программа контролирует выполнение программ пользователей для предотвращения ошибок и неправильного использования компьютера**.**

Операционные системы могут различаться особенностями реализации внутренних алгоритмов управления основными ресурсами компьютера (процессорами, памятью, устройствами), особенностями использованных методов проектирования, типами аппаратных платформ, областями использования и многими другими свойствами.

Описать операционную систему можно только путем деления ее на меньшие компоненты. Не все ОС имеют одинаковую структуру. Однако во многих современных ОС ставится следующие компоненты:

- управление процессами;
- управление основной (оперативной) памятью;
- управление вторичной (внешней) памятью;
- управление вводом-выводом;
- управление файлами;
- защита системы;
- сетевое обслуживание;
- система интерпретации команд.

ОБЩЕЕ ЗАДАНИЕ К КУРСОВОЙ РАБОТЕ

Для модели вычислительной системы (ВС) с N-ядерным процессором и мультипрограммным режимом выполнения поступающих заданий требуется разработать программную систему для имитации процесса обслуживания заданий в вычислительных системах.

При построении модели функционирования вычислительной системы должны учитываться следующие основные моменты обслуживания заданий:

- генерация нового задания;
- постановка задания в очередь для ожидания момента освобождения процессора;
- выборка задания из очереди при освобождении процессора после обслуживания очередного задания

ИНДИВИДУАЛЬНОЕ ЗАДАНИЕ (ВАРИАНТ 11)

- Метод планирования: найболее подходящий
- Вытеснение: присутствует
- Способ организации очереди: список частично упорядочивается через t тактов
- Динамическое повышение приоритета: отсутствует

РАЗДЕЛ 1 ПРОЕКТИРОВАНИЕ И ИХ РЕАЛИЗАЦИЯ

1.1 Календарное планирование проекта

Поэтапное календарное планирование для выполняемого курсового проекта представлено в табл. 1.1.

Таблица 1.1 – Календарный план курсового проектирования

№	Наименование	Содержание	Дата окончания
1	Начало	Изучение постановочного задания, предметной области	27.10.2021
2	Теоретическая подготовка	Изучение литературы по дисциплины	03.11.2021
3	Проектирование модулей проекта	Разработка компонентов программы, их функций, свойств, а также связь между ними.	10.11.2021
4	Разработка программной реализации	Написание непосредственно самой программы с использованием изученных теоретических знаний	15.11.2021
5	Тестирование	Проверка созданного приложения на предмет некорректного поведения при вводе ошибочных значений, определение уровня эргономики, устойчивости разработки	25.11.2021
5	Написание документации	Написание документации с описанием выполненной работы и трудностями с которыми пришлось встретиться во время её выполнения	29.11.2021
6	Создание презентации	Выбор наиболее интересных этапов написания программы и подготовка презентации	02.12.2021

1.1 Описание проектирования программы

1.1.1 Создается класс Configuration, отвечающий за хранение основных данных для работы программы, таких как:

- Общий объем памяти, выделяемый под фоновые процессы
- Объем памяти, занимаемый ОС
- Кол-во ядер процессора
- Минимальный и максимальный объем памяти, занимаемый процессом
- Приоритет процесса

- Минимальная и максимальная длительность процесса в тактах

- Номер блока

- Номер процесса

- Списки активных, отклоненных и оконченных процессов

Код класса:

```
public class Configuration {
     private static final int allMemory = 4096;
     public static final int OSMemory = 128;
     private static final int coresQuantity = 4;
     private static final int maxPriority = 32;
     private static final int minProcessMemory = 10;
     private static final int maxProcessMemory = 512;
     private static final int minTactInterval = 5;
     private static final int maxTactInterval = 10;
     private static final int lowerMemory = 0;
     private static boolean programWork = false;
     private static int processId = 0;
     private static int blockId = 0;

     private static String actualProcesses;
     private static String rejectedProcesses;
     private static String finalProcesses;
     private static String activeCores;
     private static String memoryBlocks;
}
```

1.1.2 Создается класс Processes, хранящий в себе данные о процессе и позволяющий вывести их на экран. Также данный класс содержит метод для сортировки процессов по их приоритету.

Код программы:

```
String processName;
    int priority;
    int id;
    int tactInterval;
    int memoryBlock;
    int startTact;
    int burstTime;
    int processCore;
    ProcessState processState;

    public static Comparator<Process> byPriority = new
Comparator<Process>() {
        @Override
        public int compare(Process o1, Process o2) {
            return o2.priority - o1.priority;
```

```
        }
    };

    @Override
    public String toString() {
        String result = "";
        result += "Name=" + processName + '\'' +
                ", Priority=" + priority +
                ", Id=" + id +
                ", MemoryBlock=" + memoryBlock +
                ", ProcessCore=" + processCore +
                ", State=" + processState;
        return result;
    }
}
```

1.1.3 Создается класс ProcessQueue, отвечающий за вывод процессов на экран и добавление процесса в список активных, отклоненных или оконченных процессов соответственно. При создании нового процесса в методе processAdd его приоритет сравнивается с первым(найболее приоритетным) процессом из списка отклоненных процессов. Процесс из данных двух, имеющий найбольший приоритет, далее будет обрабатываться программой в классе MemoryScheduler, описанный в пункте 1.1.5.. Если метод класса MemoryScheduler вернет true, то процесс добавится в список активных процессов, а также начнет выполнение на одном из ядер процессора(классы Core и CPU), если false – в список отклоненных.

Код класса:

```
public class ProcessQueue {
    private static ArrayList<Process> actualProcesses = new
ArrayList<>();
    private static ArrayList<Process> rejectedProcesses = new
ArrayList<>();
    private static ArrayList<Process> finishedProcesses = new
ArrayList<>();

    public static String finishedPrint() {
        String result = "Finished processes:\n";
        if(finishedProcesses.size() > 0) {
            for (Process process : finishedProcesses) {
                result += process + "\n";
            }
        }
        else
            result += "No processes\n";
```

```
        return result;
    }

    public static String actualPrint() {
        String result = "Actual processes:\n";
        if(actualProcesses.size() > 0) {
            for (Process process : actualProcesses) {
                result += process + "\n";
            }
        }
        else
            result += "No processes\n";
        return result;
    }

    public static String rejectedPrint() {
        String result = "Rejected processes:\n";
        if(rejectedProcesses.size() > 0) {
            for (Process process : rejectedProcesses) {
                result += process + "\n";
            }
        }
        else
            result += "No processes\n";
        return result;
    }

    public static void processAdd(int quantity) {
        for(int i = 0; i < quantity; i++)
            processAdd();
    }

    private static boolean isMaxPriority(Process process){
        if(process.priority > rejectedProcesses.get(0).priority)
            return true;
        else
            return false;
    }

    public static void processAdd() {
        Process process = new Process();
        if(rejectedProcesses.size() > 1)
            rejectedProcesses.sort(Process.byPriority);
        if(rejectedProcesses.size() > 0 &&
!isMaxPriority(process)){
            process = rejectedProcesses.get(0);
            rejectedProcesses.remove(rejectedProcesses.get(0));
        }
        if(!MemoryScheduler.fillMemoryBlock(process)) {
            process.processState = ProcessState.Waiting;
            rejectedProcesses.add(process);
        }
        else {
```

```
            process.processState = ProcessState.Running;
            process.processCore =
Utils.getRandomArrayElement(CPU.getCore().length) + 1;
            CPU.getCore()[process.processCore -
1].setProcessQuantity(
                    CPU.getCore()[process.processCore -
1].getProcessQuantity() + 1);
            CPU.getCore()[process.processCore -
1].setFree(false);
            actualProcesses.add(process);
            Configuration.memoryProcessesFill +=
process.memoryBlock;
        }
    }
}
```

1.1.4 Создание класса MemoryBlock. Данный класс содержит информацию о блоках памяти(размер, нижняя и верхняя граница, индекс содержащегося в данном блоке процесса и т.д.), а также методы, сортирующие блоки по объему памяти и верхней границе.

Код класса:

```
public class MemoryBlock {
    int startValue;
    int memoryVolume;
    int endValue;
    int idOfActiveProcess;
    int blockId;
    BlockState blockState;

    @Override
    public String toString() {
        String result = "";
        result += "BlockId=" + blockId +
                ", StartValue=" + startValue +
                ", EndValue=" + endValue +
                ", BlockState=" + blockState +
                ", ActiveProcessId=" + idOfActiveProcess;
        return result;
    }

    public static Comparator<MemoryBlock> byEnd = new
Comparator<MemoryBlock>() {
        @Override
        public int compare(MemoryBlock o1, MemoryBlock o2) {
            return o1.endValue - o2.endValue;
        }
    };
```

```
    public static Comparator<MemoryBlock> byVolume = new
Comparator<MemoryBlock>() {
        @Override
        public int compare(MemoryBlock o1, MemoryBlock o2) {
            return o1.memoryVolume- o2.memoryVolume;
        }
    };
}
```

1.1.5 Создание класса MemoryScheduler. Данный класс отвечает за основной функционал всей программы. При попытке добавить новый процесс он будет передан как параметр в метод fillMemoryBlock, который в свою очередь сортирует блоки по объему памяти, после чего в первый, найболее подходящий по размеру свободный блок записывает наш процесс. Далее, если свободный блок был найден, блоки снова сортируются в методе afterMemoryFill по окончанию, после чего Id процесса записывается в данный блок. В конце выполнения функции создается новый, свободный для дальнейшего использования блок памяти. Данный класс также содержит метод release, вызывающийся каждую секунду из класса ProgramTimer. Данный метод сортирует список активных процессов по окончанию, увеличивая после этого такт каждого из процессов на секунду соответственно. Если количество тактов процесса равно его диапазону, в котором тот должен отработать, то процесс должен завершить свою работу. Процесс добавляется в список завершенных, блок памяти, в котором тот содержался, очищается, и если рядом с данным блоком присутствуют еще свободные, то они объединяются в один блок.

Код класса:

```
public class MemoryScheduler {
    private static ArrayList<MemoryBlock> memoryBlocks = new
ArrayList<>();

    public static String print() {
        String result = "Memory blocks:\n";
        for (MemoryBlock memoryBlock : memoryBlocks) {
            result += memoryBlock + "\n";
        }
        return result;
    }
```

```
    public static void blockAdd(MemoryBlock memoryBlock){
        memoryBlocks.add(memoryBlock);
    }

    private static void afterMemoryFill(int blockId) {
        if(memoryBlocks.size() > 1)
            memoryBlocks.sort(MemoryBlock.byEnd);
        for(int j = 0; j < memoryBlocks.size(); j++) {
            if (memoryBlocks.get(j).blockId == blockId) {
                if (j != memoryBlocks.size() - 1) {
                    memoryBlocks.get(j + 1).startValue =
memoryBlocks.get(j).endValue;
                    memoryBlocks.get(j + 1).memoryVolume =
                            memoryBlocks.get(j + 1).endValue -
memoryBlocks.get(j + 1).startValue;
                } else
                    blockAdd(new
MemoryBlock(memoryBlocks.get(j).endValue,
Configuration.getAllMemory()));
            }
        }
    }

    public static boolean fillMemoryBlock(Process process) {
        int blockId;
        if(memoryBlocks.size() > 1)
            memoryBlocks.sort(MemoryBlock.byVolume);
        for(int i = 0; i < memoryBlocks.size(); i++) {
            if(memoryBlocks.get(i).blockState ==
BlockState.Empty) {
                if(memoryBlocks.get(i).memoryVolume >
process.memoryBlock) {
                    memoryBlocks.get(i).blockState =
BlockState.Fill;
                    memoryBlocks.get(i).idOfActiveProcess =
process.id;
                    memoryBlocks.get(i).memoryVolume =
process.memoryBlock;
                    memoryBlocks.get(i).endValue =
memoryBlocks.get(i).startValue +
memoryBlocks.get(i).memoryVolume;
                    blockId = memoryBlocks.get(i).blockId;
                    afterMemoryFill(blockId);
                    return true;
                }
            }
        }
        return false;
    }

    private static void blockBefore(int index){
```

```
        memoryBlocks.get(index).startValue -=
memoryBlocks.get(index - 1).memoryVolume;
        memoryBlocks.get(index).memoryVolume +=
memoryBlocks.get(index - 1).memoryVolume;
        memoryBlocks.remove(memoryBlocks.get(index - 1));
    }

    private static void blockAfter(int index){
        memoryBlocks.get(index).endValue +=
memoryBlocks.get(index + 1).memoryVolume;
        memoryBlocks.get(index).memoryVolume +=
memoryBlocks.get(index + 1).memoryVolume;
        memoryBlocks.remove(memoryBlocks.get(index + 1));
    }

    private static void freePastBlock(int index, int
freeBlocksQuantity){
        if(index == memoryBlocks.size() - 1 &&
memoryBlocks.get(index).startValue == memoryBlocks.get(index -
1).endValue) {
            if(memoryBlocks.get(index - 1).blockState ==
BlockState.Empty) {
                blockBefore(index);
                freeBlocksQuantity++;
            }
        }
    }

    private static void freeNextBlock(int index, int
freeBlocksQuantity){
        if (index == 0 && memoryBlocks.get(index).endValue ==
memoryBlocks.get(index + 1).startValue) {
            if(memoryBlocks.get(index + 1).blockState ==
BlockState.Empty) {
                blockAfter(index);
                freeBlocksQuantity++;
            }
        }
    }

    private static void freeBothBlocks(int index, int
freeBlocksQuantity){
        if(index > 0 && index < memoryBlocks.size() - 1) {
            if(memoryBlocks.get(index).endValue ==
memoryBlocks.get(index + 1).startValue &&
                    memoryBlocks.get(index).startValue ==
memoryBlocks.get(index - 1).endValue) {
                if (memoryBlocks.get(index + 1).blockState ==
BlockState.Empty) {
                    blockAfter(index);
                    freeBlocksQuantity++;
                }
```

```
                if (memoryBlocks.get(index - 1).blockState ==
BlockState.Empty) {
                    blockBefore(index);
                    freeBlocksQuantity++;
                }
            }
        }
    }

    public static void release() {
        int freeBlocksQuantity = 0;
        if(ProcessQueue.getActualProcesses().size() > 0) {
            memoryBlocks.sort(MemoryBlock.byEnd);
            for (int i = 0; i <
ProcessQueue.getActualProcesses().size(); i++) {

ProcessQueue.getActualProcesses().get(i).burstTime++;
                if
(ProcessQueue.getActualProcesses().get(i).burstTime ==
ProcessQueue.getActualProcesses().get(i).tactInterval) {
                    for(int j = 0; j < memoryBlocks.size(); j++)
{
                        if(memoryBlocks.get(j).idOfActiveProcess
== ProcessQueue.getActualProcesses().get(i).id) {
                            memoryBlocks.get(j).blockState =
BlockState.Empty;

memoryBlocks.get(j).idOfActiveProcess = -1;
                            if(memoryBlocks.size() > 1) {
                                freeNextBlock(j,
freeBlocksQuantity);
                                freePastBlock(j,
freeBlocksQuantity);
                                freeBothBlocks(j,
freeBlocksQuantity);
                            }

CPU.getCore()[ProcessQueue.getActualProcesses().get(i).processCo
re - 1].setProcessQuantity(

CPU.getCore()[ProcessQueue.getActualProcesses().get(i).processCo
re - 1].getProcessQuantity() - 1);

if(CPU.getCore()[ProcessQueue.getActualProcesses().get(i).proces
sCore - 1].getProcessQuantity() == 0)

CPU.getCore()[ProcessQueue.getActualProcesses().get(i).processCo
re - 1].setFree(true);
                            Configuration.memoryProcessesFill -=
ProcessQueue.getActualProcesses().get(i).memoryBlock;

ProcessQueue.getActualProcesses().get(i).processState =
ProcessState.Finished;
```

```
ProcessQueue.getActualProcesses().get(i).processCore = -1;

ProcessQueue.getFinishedProcesses().add(ProcessQueue.getActualPr
ocesses().get(i));

ProcessQueue.getActualProcesses().remove(ProcessQueue.getActualP
rocesses().get(i));
                        }
                    }
                }
            }
        }

    public static ArrayList<MemoryBlock> getMemoryBlocks() {
        return memoryBlocks;
    }
}
```

1.1.6 Создание класса ProgramTimer. По названию можно понять, что данный класс реализует таймер, увеличивающий такт работы программы на секунду и вызывающий метод release класса MemoryScheduler. Метод по увеличению такта выполняется в отдельном потоке.

Код класса:

```
public class ProgramTimer {
    private static int programTact = 0;

    Timer timer = new Timer();
    TimerTask timerTask = new TimerTask() {
        @Override
        public void run() {
            incTact();
        }
    };

    public static int getProgramTact() { return programTact; }

    public static void incTact() {
        programTact++;
        MemoryScheduler.release();
    }

    public void start() {
        timer.scheduleAtFixedRate(timerTask, 1000, 1000);
    }
}
```

1.1.7 Создание класса CPU. Класс хранит в себе массив ядер типа Core, а также метод для инициализации объектов, хранящихся в данном массиве.

Код класса:

```
public class CPU {
    private static Cores[] core;

    public static String corePrint() {
        String result = "Cores busyness: ";
        for (Cores cores : core) {
            result += cores + " ";
        }
        return result;
    }

    public static void coresInitialization(){
        core = new Cores[Configuration.getCoresQuantity()];
        for(int i = 0; i < core.length; i++)
            core[i] = new Cores();
    }
}
```

1.1.8 Создание класса Core. Хранит в себе информацию о ядрах процессора.

Код класса:

```
public class Cores {
    private int processQuantity;
    private boolean isFree;

    @Override
    public String toString() {
        String result = "";
        result += isFree + " ; ";
        return result;
    }

    public Cores() {
        this.processQuantity = 0;
        this.isFree = true;
    }
}
```

1.1.9 Создание класса Scheduler. Метод данного класса programStart вызывается в самом начале работы программы. В нем вызывается метод для инициализации ядер процессора, запускается таймер и добавляется первый блок памяти. Также в данном классе присутствует метод, выполняющийся в

отдельном потоке и каждые пять секунд добавляющий два процесса в наши блоки памяти.

Код класса:

```
public class Scheduler extends Thread {

    public static void programStart() {
        CPU.coresInitialization();
        Configuration.setProgramWork(true);
        ProgramTimer timer = new ProgramTimer();
        timer.start();
        MemoryScheduler.blockAdd(new
MemoryBlock(Configuration.getLowerMemory(),
Configuration.getAllMemory()));
    }

    @Override
    public void run(){
        while (Configuration.isProgramWork()){
            ProcessQueue.processAdd(2);

Configuration.setActualProcesses(ProcessQueue.actualPrint());

Configuration.setRejectedProcesses(ProcessQueue.rejectedPrint())
;

Configuration.setFinalProcesses(ProcessQueue.finishedPrint());

Configuration.setMemoryBlocks(MemoryScheduler.print());
            Configuration.setActiveCores(CPU.corePrint());
            try {
                Thread.sleep(Configuration.threadSleep);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

1.1.10 Добавление интерфейса, создание класса Controller. Класс представляет собой визуальную часть нашей прогрыммы. При нажатии кнопки Start программа запускает таймер, поток добавления процессов и поток вывода данных на экран каждые пять секунд. Кнопка End завершает выполнение программы.

Код класса:

```
public class Controller extends Thread {

    private Stage stage = new Stage();
    private Scheduler scheduler = new Scheduler();

    @FXML
    private ResourceBundle resources;

    @FXML
    private URL location;

    @FXML
    private TextArea actualProcessField = new TextArea();

    @FXML
    private TextArea busyBlocksField = new TextArea();

    @FXML
    private TextArea busyCoresField = new TextArea();

    @FXML
    private Button exitBtn;

    @FXML
    private TextArea finishedProcessField = new TextArea();

    @FXML
    private TextArea rejectedProcessField = new TextArea();

    @FXML
    private Button startProgramBtn;

    @Override
    public void run(){
        while(Configuration.isProgramWork()) {

actualProcessField.setText(Configuration.getActualProcesses());

rejectedProcessField.setText(Configuration.getRejectedProcesses(
));

finishedProcessField.setText(Configuration.getFinalProcesses());

busyBlocksField.setText(Configuration.getMemoryBlocks());

busyCoresField.setText(Configuration.getActiveCores());
            try {
                Thread.sleep(Configuration.threadSleep);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
```

```
        }
    }

    @FXML
    void initialize() {
        startProgramBtn.setOnAction(event -> {
            if(!Configuration.isProgramWork()) {
                Scheduler.programStart();
                scheduler.start();
                start();
            }
        });
        exitBtn.setOnAction(event -> {
            if(Configuration.isProgramWork())
                Configuration.setProgramWork(false);

((Stage)(((Button)event.getSource()).getScene().getWindow())).cl
ose();
            stage.close();
            System.exit(0);
        });
    }
}
```

1.1.11 Подключенные утилиты

Для написания данной программы необходимо подключить элемент коллекции ArrayList, интерфейс Comparator и класс Timer. Пакеты для работы JavaFX программа подключает автоматически:

```
import java.net.URL;
import java.util.ResourceBundle;
import javafx.fxml.FXML;
import javafx.scene.control.Button;
import javafx.scene.control.TextArea;
import javafx.stage.Stage;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.Timer;
import java.util.TimerTask;
```

РАЗДЕЛ 2 СПРАВОЧНАЯ ИНФОРМАЦИЯ ДЛЯ ПОЛЬЗОВАТЕЛЯ

1.1 Краткое описание продукта

Приложение представляет собой ПО, эмулирующее работу диспетчера задач. Функциональность программы включает:

- Добавление и удаление блоков памяти для хранения выполняющихся процессов.
- Добавление процессов в список активных если блок памяти необходимого размера присутствует, в список ожидания если подходящий блок памяти отсутствует на момент попытки добавления, в список завершенных процессов если время выполнения процесса подошло к концу.
- Работу таймера для выполнения программного кода в автоматическом режиме.
- Эмуляцию загруженности ядер процессора. Если ядро использует хотя-бы один процесс – ядро занято.
- Вывод информации о блоках памяти, а также данные обо всех процессах.

1.2 Работа программы

При открытии эмулятора пользователю необходимо лишь нажать на кнопку Start, после чего программа начнет выполнение в автоматическом режиме:

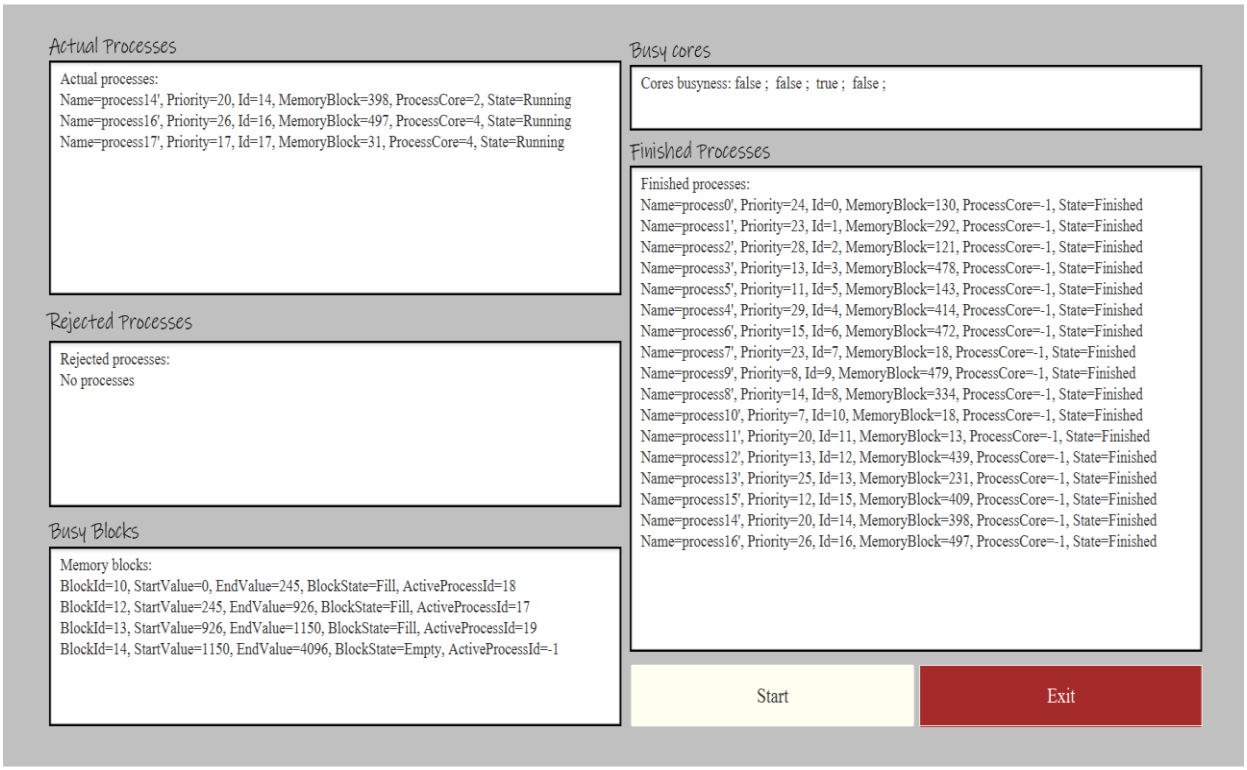

Рисунок 2.1 – Пример работы программы в автоматическом режиме

ВЫВОД

На основании знаний, приобретенных на курсе «Сучасні технології програмування», я создал приложение, эмулирующее работу диспетчера задач. Через каждый заданный помежуток времени программа генерирует новые процессы, выполняемые нашей ОС, и добавляет их в отведенные для этого доступные блоки памяти. На каждом такте выполнения программы проводится увеличение времени работы каждого из процессов. После окончания работы процесса соответствующий блок памяти очищается и становится пригодным для дальнейшего использования другими процессами. Также важно отметить, что каждый их процессов выполняется на конкретном ядре процессора, в следствии чего мы можем понять загружено ядро или свободно.

Для создание пользовательского интерфейса была использована платформа JavaFX.

Данная работа принесла мне большой опыт написания кода с использованием JavaFX и на языке Java в целом.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Фрэд Лонг, Дхрув Мохиндра, Роберт С. Сикорд, Дин Ф. Сазерленд, Дэвид Свобода. Руководство для программиста на Java. - М.: Вильямс, 2014. - 256 с
2. Джошуа Блох. Java. Эффективное программирование. - М.: Диалектика, 2019. — 464 с.
3. Бенджамин Дж. Эванс, Джеймс Гоф, Крис Ньюленд. Java: оптимизация программ. Практические методы повышения производительности приложений. — М.: Диалектика, 2019. — 448 с