Одесский Национальный Политехнический Университет

Кафедра «Компьютеризированные системы управления»

**КУРСОВАЯ РОБОТА**

по дисциплине «Современные технологии программирования»

на тему: «Эмуляция работы операционной системы»

Вариант 6

Студента 3 курса, группы АТ–183

специальности «Автоматизация и компьютерно-интегрированные технологии»

Левдиков М. А.

Руководитель: доц. Сперанский В.А.

Национальная шкала:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Количество баллов:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Оценка: ECTS\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Члены комиссии \_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись) (фамилия и инициалы)

\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись) (фамилия и инициалы)

\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись) (фамилия и инициалы)

**г. Одесса – 2020 год**

Содержание

[**ВВЕДЕНИЕ** 3](#_Toc58706981)

[**РАЗРАБОТКА ПРИЛОЖЕНИЯ** 5](#_Toc58706982)

[**Вариант задания** 5](#_Toc58706983)

[**Постановка задачи** 5](#_Toc58706984)

[**Описание алгоритмов планирования процессора и памяти** 5](#_Toc58706985)

[**Алгоритмы планирования процессов** 6](#_Toc58706986)

[**Алгоритм FCFS** 6](#_Toc58706987)

[**Алгоритм RR** 6](#_Toc58706988)

[**Алгоритм SJF** 7](#_Toc58706989)

[**Приоритетное планирование** 7](#_Toc58706990)

[**Алгоритм SRR** 8](#_Toc58706991)

[**Алгоритмы планирования памяти** 9](#_Toc58706992)

[**Page replacement algorithm** 9](#_Toc58706993)

[**Алгоритм FIFO** 10](#_Toc58706994)

[**Алгоритм LRU (Least Recently Used)** 10](#_Toc58706995)

[**Алгоритм буферизации станиц** 11](#_Toc58706996)

[**Алгоритм LFU (Least Frequently Used)** 11](#_Toc58706997)

[**Диаграмма движения процессов** 11](#_Toc58706998)

[**Объектная модель** 12](#_Toc58706999)

[**1. Process:** 13](#_Toc58707000)

[**2. Queue:** 14](#_Toc58707001)

[**3. Scheduler:** 14](#_Toc58707002)

[**4. Configuration:** 15](#_Toc58707003)

[**5. MemoryBlock:** 15](#_Toc58707004)

[**6. MemoryScheduler:** 15](#_Toc58707005)

[**7. Resource:** 16](#_Toc58707006)

[**8. Device:** 16](#_Toc58707007)

[**9. CPU:** 16](#_Toc58707008)

[**10. Core (наследник класса Resource)** 17](#_Toc58707009)

[**11. Core (наследник класса Resource)** 17](#_Toc58707010)

[**12. Utils** 17](#_Toc58707011)

[**13. Controller** 17](#_Toc58707012)

[**Исходные текста программы** 18](#_Toc58707013)

[**ИНСТРУКЦИЯ** 37](#_Toc58707014)

[**СПИСОК ЛИТЕРАТУРЫ** 38](#_Toc58707015)

# **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.

ОС реализует множество различных функций, в том числе:

* определяет так называемый интерфейс пользователя;
* обеспечивает разделение аппаратных ресурсов между пользователями;
* дает возможность работать с общими данными в режиме коллективного пользования;
* планирует доступ пользователя к общим ресурсам;
* обеспечивает эффективное выполнение операций ввода-вывода;
* осуществляет восстановление информации и вычислительного процесса в случае ошибок.

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.

В зависимости от особенностей использованного алгоритма управления процессором, операционные системы делят на многозадачные и однозадачные, многопользовательские и однопользовательские, на системы, поддерживающие многопоточную обработку и не поддерживающие ее, на многопроцессорные и однопроцессорные системы.

Важное влияние на облик операционной системы в целом, на возможности ее использования в той или иной области оказывают особенности и других подсистем управления локальными ресурсами  подсистем управления памятью, файлами, устройствами ввода-вывода.

Специфика ОС проявляется и в том, каким образом она реализует сетевые функции: распознавание и перенаправление в сеть запросов к удаленным ресурсам, передача сообщений по сети, выполнение удаленных запросов.

Многозадачные ОС подразделяются на три типа в соответствии с использованными при их разработке критериями эффективности:

* системы пакетной обработки (например, OC EC),
* системы разделения времени (UNIX, VMS),
* системы реального времени (QNX, RT/11).

Некоторые операционные системы могут совмещать в себе свойства систем разных типов, например, часть заданий может выполняться в режиме пакетной обработки, а часть  в режиме реального времени или в режиме разделения времени. В таких случаях режим пакетной обработки часто называют фоновым режимом.

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;

Оптимизация работы ОС с процессами напрямую влияет на быстродействие обработки данных.

# **РАЗРАБОТКА ПРИЛОЖЕНИЯ**

## **Вариант задания**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| n/n |  | Стратегия планирования | Наличие вытеснения | Способ организации очереди | | | | | Динамическое повышение приоритета | | Критерий вытеснения для SJF | |
|  | упорядоченный  список | не упорядочен.  список | список частично упорядочивается  через t тактов | | каждому  приоритету своя очередь |
| метод планирования памяти | По интервалу непрерывного выполнения | По оставшемуся времени |
| 6. | 3 | HPF | - |  | + |  |  | | | + |  |  |

## **Постановка задачи**

Целью курсовой работы является изучение основных методов, используемых при управлении ресурсами в различных операционных системах.

Задачей курсовой работы является получение, как теоретических знаний, так и практических навыков, достаточных для проектирования и программирования системного программного обеспечения современных компьютеров, ознакомление с проблемами моделирования и анализа эффективности функционирования реальных вычислительных систем.

# **Описание алгоритмов планирования процессора и памяти**

**Планирование ЦП** — это процесс определения того, какому процессу будет принадлежать ЦП для выполнения, пока другой процесс находится в режиме ожидания. Основная задача планирования ЦП состоит в том, чтобы гарантировать, что всякий раз, когда ЦП остается бездействующим, ОС, по крайней мере, выбирает один из процессов, доступных в готовой очереди для выполнения. Процесс выбора будет выполняться планировщиком ЦП. Он выбирает один из процессов в памяти, которые готовы к выполнению.

## **Алгоритмы планирования процессов**

### **Алгоритм FCFS**

FCFS (first come – first serve; первым пришел – первым обслуживается) – простейший алгоритм, работа, которой понятна из ее названия. Это алгоритм без вытеснения, то есть процесс, выбранный для выполнения на ЦП, не прерывается, пока не завершится (или не перейдет в состояние ожидания по собственной инициативе). FCFS обеспечивает минимум накладных расходов. Среднее потерянное время при применении этого алгоритма не зависит от длительности процесса, но штрафное отношение при равном потерянном времени будет большим для коротких процессов. Поэтому алгоритм FCFS считается лучшим для длинных процессов. Существенным достоинством этого алгоритма наряду с его простотой является то обстоятельство, что FCFS гарантирует отсутствие бесконечного откладывания процессов: любой поступивший в систему процесс будет, в конце концов, выполнен независимо от степени загрузки системы.

### **Алгоритм RR**

RR (round robin – карусель) – простейший алгоритм с вытеснением. Процесс получает в свое распоряжение ЦП на некоторый квант времени Q (в простейшем случае размер кванта фиксирован). Если за время Q процесс не завершился, он вытесняется из ЦП и направляется в конец очереди готовых процессов, где ждет выделения ему следующего кванта, и т.д. Показатели эффективности RR существенно зависят от выбора величины кванта Q. RR обеспечивает наилучшие показатели, если длительность большинства процессов приближается к размеру кванта, но не превосходит его. Тогда большинство процессов укладываются в один квант и не становятся в очередь повторно. При величине кванта, стремящейся к бесконечности, RR вырождается в FCFS. При Q, стремящемся к 0, накладные расходы на переключение процессов возрастают настолько, что поглощают весь ресурс ЦП. RR обеспечивает наилучшие показатели справедливости: штрафное отношение P на большом участке длительностей процессов t остается практически постоянным. Только на участке t

### **Алгоритм SJF**

SJF (shortest job first – самая короткая работа – первой) – невытесняющий алгоритм, в котором наивысший приоритет имеет самый короткий процесс. Для того чтобы применять этот алгоритм, должна быть известна длительность процесса – задаваться пользователем или вычисляться методом экстраполяции. Для коротких процессов SJN обеспечивает лучшие показатели, чем RR, как по потерянному времени, так и по штрафному отношению. SJN обеспечивает максимальную пропускную способность системы – выполнение максимального числа процессов в единицу времени, но показатели для длинных процессов значительно худшие, а при высокой степени загрузки системы активизация длинных процессов может откладываться до бесконечности. Штрафное отношение слабо изменяется на основном интервале значений t, но значительно возрастает для самых коротких процессов: такой процесс при поступлении в систему имеет самый высокий приоритет, но вынужден ждать, пока закончится текущий активный процесс.

### **Приоритетное планирование**

Алгоритм SJF представляет собой частный случай приоритетного планирования. При приоритетном планировании каждому процессу присваивается определенное числовое значение - приоритет, в соответствии с которым ему выделяется процессор. Процессы с одинаковыми приоритетами планируются в порядке FCFS. Для алгоритма SJF в качестве такого приоритета выступает оценка продолжительности следующего CPU burst. Чем меньше значение этой оценки, тем более высокий приоритет имеет процесс. Принципы назначения приоритетов могут опираться как на внутренние критерии вычислительной системы, так и на внешние по отношению к ней. Внутренние используют различные количественные и качественные характеристики процесса для вычисления его приоритета. Это могут быть, например, определенные ограничения по времени использования процессора, требования к размеру памяти, число открытых файлов и используемых устройств ввода-вывода, отношение средних продолжительностей I/O burst к CPU burst и т. д. Внешние критерии исходят из таких параметров, как важность процесса для достижения каких-либо целей, стоимость оплаченного процессорного времени и других политических факторов. Планирование с использованием приоритетов может быть как вытесняющим, так и невытесняющим. При вытесняющем планировании процесс с более высоким приоритетом, появившийся в очереди готовых процессов, вытесняет исполняющийся процесс с более низким приоритетом. В случае невытесняющего планирования он просто становится в начало очереди готовых процессов.

### **Алгоритм SRR**

SRR (selfish RR – эгоистичный RR) – метод с вытеснением, дающий дополнительные преимущества выполняемым процессам, что позволяет повысить пропускную способность. Все процессы разделяются на две категории: новые и выбранные. Новыми считаются те процессы, которые не получили еще ни одного кванта времени ЦП, все остальные процессы – выбранные. При поступлении в систему каждому процессу дается некоторый приоритет P0, одинаковый для всех процессов, который в дальнейшем возрастает. В конце каждого кванта времени пересчитываются приоритеты всех процессов, причем приоритеты новых процессов возрастают на величину dA, а выбранных – на величину dB. ЦП отдается процессу с наивысшим приоритетом, а при равенстве приоритетов – тому, который раньше поставлен в очередь. Показатели алгоритма существенно зависят от выбранного соотношения между dA и dB. При dB/dA=0 алгоритм вырождается в обыкновенный RR, при dB/dA>=1 – в FCFS. Собственно, алгоритм SRR обеспечивается в диапазоне значений 0<1.

## **Алгоритмы планирования памяти**

### **Page replacement algorithm**

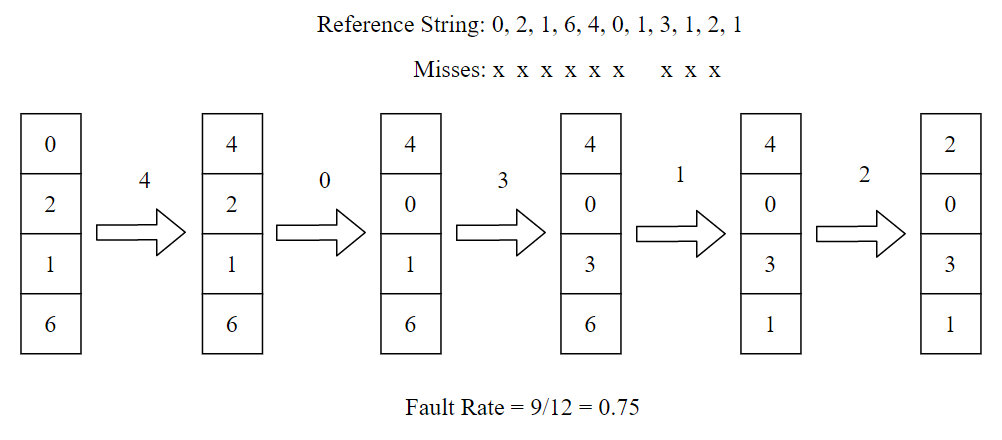
Page replacement algorithms (алгоритмы замены страниц) – это методы, с помощью которых операционная система решает, какие страницы памяти следует выгружать, записывать на диск, когда необходимо выделить страницу памяти. Пейджинг происходит всякий раз, когда происходит сбой страницы, и свободная страница не может использоваться для учета целей распределения по причине того, что страницы недоступны или количество свободных страниц меньше, чем требуемых страниц.

Когда страница, которая была выбрана для замены и была выгружена, снова обращается к ней, она должна считываться с диска, а это требует завершения ввода-вывода. Этот процесс определяет качество алгоритма замены страницы: чем меньше время ожидания страниц, тем лучше алгоритм.

Алгоритм замены страниц смотрит на ограниченную информацию о доступе к страницам, предоставляемую оборудованием, и пытается выбрать, какие страницы следует заменить, чтобы минимизировать общее количество пропусков страниц, при этом уравновешивая это с затратами на первичное хранилище и процессорное время алгоритма. сам. Есть много разных алгоритмов замены страниц. Мы оцениваем алгоритм, выполняя его для конкретной строки обращения к памяти и вычисляя количество ошибок страниц.

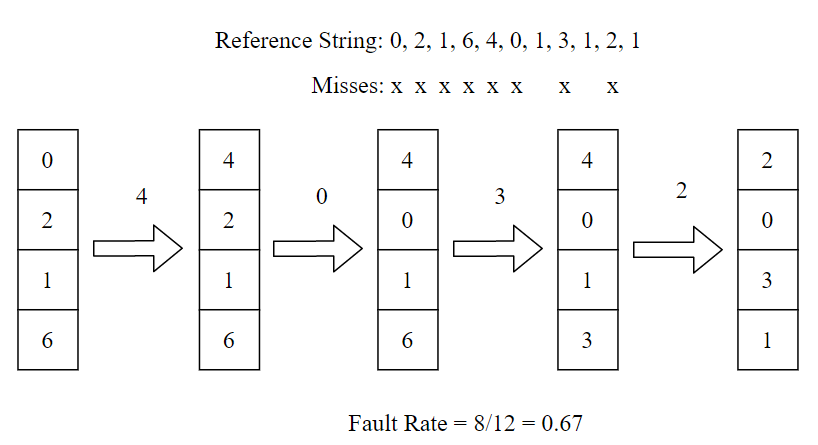
### **Алгоритм FIFO**

* Самая старая страница в основной памяти - это та, которая будет выбрана для замены.
* Легко реализовать, вести список, заменять страницы из хвоста и добавлять новые страницы в голове.



### **Алгоритм LRU (Least Recently Used)**

* Cтраница, которая долгое время не использовалась в основной памяти, будет выбрана для замены.
* Легко реализовать, вести список, заменять страницы, глядя в прошлое.



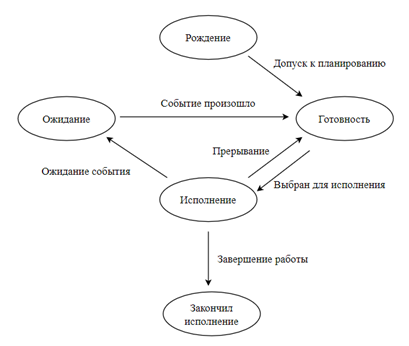
### **Алгоритм буферизации станиц**

* Чтобы процесс начался быстро, держите пул свободных кадров.
* В случае ошибки страницы выберите страницу для замены.
* Напишите новую страницу в рамке свободного пула, отметьте таблицу страниц и перезапустите процесс.
* Теперь запишите грязную страницу с диска и поместите фрейм, содержащий замененную страницу, в свободный пул.

**Алгоритм LFU (Least Frequently Used)**

* Страница с наименьшим счетчиком будет выбрана для замены.
* Этот алгоритм страдает от ситуации, когда страница интенсивно используется на начальном этапе процесса, но затем больше не используется.

**Диаграмма движения процессов**



**Объектная модель**

**Классы:**

1. Process
2. Queue
3. Scheduler
4. Configuration
5. MemoryBlock
6. MemoryScheduler
7. Resource
8. Device
9. CPU
10. Core
11. TactGenerator
12. Utils
13. Controller

**1. Process:**

Поля:

* id
* priority
* time (число, сколько должен отработать процесс)
* memory (занимаемая процессом память)
* arrivalTime (время, когда процесс поступил в систему)
* burstTime
* idleTime (кол-во тактов тактового генератора, которое проводит процесс в состояние waiting; поле необходимое для корректного вычисления bustTime)
* runTime (такт, с которого процесс переходит в Running, используется для вычисления burtTime)
* state (состояние процесса)
* memoryBlock
* core (ссылка на ядро, на которм выполняется процесс во время выполнения)
* device (инкапсулирует ссылку на устройство (device), к которому должен обращаться процесс)
* resourceId (идентификатор ресурса, который занимает процесс во время обращения к устройству. Необходим для мониторинга занятости ресурса.)
* timer (таймер, используемый для симуляции обращения к устройству)

Методы:

* Конструктор (устанавливает начальное значение параметров процесса)
* Геттеры и сеттеры для различных полей.
* startTimer (запуск таймера симуляции обращения к устройству. Таймер запускается однократно во время выполнения процесса через случайный интервал, равный половине периода тактового генератора)
* stopTimer (останавливает таймер.

**2. Queue:**

Поля:

* queue (ArrayList процессов)
* lastID

Методы:

* Конструктор
* Геттеры, сеттеры
* add (перегружен)
* remove (перегружен)
* sortByPriorityAndArrivalTime (сортирует процессы в очереди по времени прибытия и приоритету процесса)
* getCountByState (производит подсчет процессов в очереди с определенным состоянием процесса)

**3. Scheduler:**

Поля:

* jobsQueue
* readyQueue
* waitingQueue
* rejectedQueue
* memoryScheduler
* tactGenerator
* cpu
* device

Методы:

* Конструктор
* Геттеры, сеттеры
* addProcess
* addProcessRandom (добавляет процесс в случайный момент времени)
* init (запускает тактовый генератор и производит создание блоков памяти в режиме симуляции)
* schedule (реализация диспетчера посредством создания бесконечного цикла с задержкой, равной половине периода тактового генератора, в каждом проходе цикла производится обработка текущих процессов и управление очередями процессов);
* initProcess (служебный метод, посредством которого производится попытка выделения процессу блока памяти с последующим переводом его в состояние Ready. В случае отсутствия возможности выделения памяти, процесс удаляется из jobsQueue и помещается в rejectedQueue).

**4. Configuration:**

Поля:

* memoryVolume;
* maxPriority.

**5. MemoryBlock:**

Поля:

* availableMemory;
* Компаратор.

Методы:

* Конструктор;
* Геттер, сеттер.

**6. MemoryScheduler:**

Поля:

* memoryBlocks (ArrayList блоков памяти);
* Компаратор.

Методы:

* fillMemoryBlock (Выбирает неподходящий блок (самый большой из доступных) и вставляет туда процесс (отнимает значение занимаемой памяти процессом от доступной памяти блока);

**7. Resource:**

Поля:

* isIdle;
* resourceNumber;

Методы:

* Конструктор
* getNumber;
* Сеттер и геттер для поля isIdle;

**8. Device:**

Поля:

* Resources;

Методы:

* Конструктор
* Геттер;
* getResource (возвращает один ресурс из массива ресурсов)
* requestResource (ищет свободный ресурс и возвращает его дескриптор, в случае отсутствия свободного ресурса, возвращает -1. Метод реализован как synchronized для предотвращения одновременного доступа к ресурсу из разных потоков);
* doWork (метод симулирует обработку процесса ресурсом в течении случайного интервала времени в диапазоне от 10 до 20 секунд).

**9. CPU:**

Поля:

* cores;

Методы:

* Конструктор
* Геттер и сеттер для поля cores;
* getCore (возвращает один core из массива)

**10. Core (наследник класса Resource)**

Методы:

* Конструктор.

**11. Core (наследник класса Resource)**

Методы:

* Конструктор.

**12. Utils**

Поля:

* random.

Методы:

* getRandomInteger (перегружен).

**13. Controller**

Поля:

* Поля таблиц и столбцов таблиц;
* main;
* processObservableList;
* memoryObservableList;
* cpuObservableList;
* resourceObservableList.

Методы:

* initialize (метод, необходимый для того, чтобы иметь возможность настраивать компоненты);
* addNumberOfProcessesButton;
* addNumberOfProcessesInTandomTimeButton;
* killProcess.

**Исходные текста программы**

1. листинг кода класса Process:

package com.company.classes;  
  
import java.util.Date;  
import java.util.Timer;  
import java.util.TimerTask;  
  
public class Process{  
 private int id;  
 private String name;  
 private int priority;  
 private int time; private int memory; private int arrivalTime; private int burstTime; private int idleTime;  
 private int runTime;private State state;  
 private MemoryBlock memoryBlock;  
 private Core core;  
 private Device device;  
 private int resourceId = -1;  
 private Timer timer = new Timer();  
public Process(int id) {  
 this.id = id;  
 this.memory = Utils.*getRandomInteger*(10,Configuration.*memoryVolume*/4); this.priority = Utils.*getRandomInteger*( Configuration.*maxPriority*);  
 this.time = Utils.*getRandomInteger*(10,100);  
 this.arrivalTime = TactGenerator.*getTime*();  
 this.burstTime = 0;  
 this.name = "PID" + this.id;  
 this.state = State.*New*;  
 }  
public int getId() {  
 return id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public int getPriority() {  
 return priority;  
 }  
  
 public int getTime() {  
 return time;  
 }  
  
 public int getMemory() {  
 return memory;  
 }  
  
 public int getArrivalTime() {  
 return arrivalTime;  
 }  
  
 public int getBurstTime() {  
 return burstTime;  
 }  
  
 public State getState() {  
 return state;  
 }  
  
 public void setArrivalTime(int arrivalTime) {  
 this.arrivalTime = arrivalTime;  
 }  
  
 public void setBurstTime(int burstTime) {  
 this.burstTime = burstTime;  
 }  
  
 public void setState(State state) {  
 this.state = state;  
 if(state == State.*Running*){  
 startTimer();  
 }  
 else if(state == State.*Terminated*){  
 stopTimer();  
 core.setIdle(true);  
 core = null;  
 releaseMemory();  
 }  
 }  
  
 public void setPriority(int priority) {  
 this.priority = priority;  
 }  
  
 public Core getCore() {  
 return core;  
 }  
  
 public void setCore(Core core) {  
 this.core = core;  
 }  
  
 public MemoryBlock getMemoryBlock() {  
 return memoryBlock;  
 }  
  
 public void setMemoryBlock(MemoryBlock memoryBlock) {  
 this.memoryBlock = memoryBlock;  
 }  
  
 public void releaseMemory(){  
 int memory = memoryBlock.getAvailableMemory();  
 this.memoryBlock.setAvailableMemory(memory += this.memory);  
 }  
  
 public void setDevice(Device device) {  
 this.device = device;  
 }  
public void setId(int id) {  
 this.id = id;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public void setTime(int time) {  
 this.time = time;  
 }  
  
 public void setMemory(int memory) {  
 this.memory = memory;  
 }  
  
 public int getIdleTime() {  
 return idleTime;  
 }  
  
 public int getRunTime() {  
 return runTime;  
 }  
  
 public void setRunTime(int runTime) {  
 this.runTime = runTime;  
 }  
  
 public int getResourceId() {  
 return resourceId;  
 }  
  
 private void startTimer(){  
 int requestDeviceTime = Utils.*getRandomInteger*(0,time / 2);  
 var me = this;  
  
 TimerTask repeatedTask = new TimerTask() {  
 int startWaitTime = 0;  
  
 public void run() {  
if(burstTime >= requestDeviceTime && state == State.*Running*){  
 System.*out*.println("@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@");  
 System.*out*.println(device);  
 System.*out*.println("@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@");  
 state = State.*Waiting*;  
 startWaitTime = TactGenerator.*getTime*();  
 int resourceNumber = device.requestResource();  
 if(resourceNumber != -1){  
 device.doWork(resourceNumber);  
 idleTime += (TactGenerator.*getTime*() - startWaitTime);  
 state = State. *Running*;  
 stopTimer();  
 }  
 else  
 System.*out*.println(new Date() + " Process " + me.name + " is waiting for resource");  
 }  
  
 }  
 };  
  
 timer.schedule(repeatedTask, 500, 500);  
 }  
 private void stopTimer(){  
 timer.cancel();  
 }  
  
 @Override  
 public String toString() {  
 return id +  
 "{ PID='" + name + '\'' +  
 ", priority=" + priority +  
 ", time=" + time +  
 ", memory=" + memory +  
 ", timeIn=" + arrivalTime +  
 ", burstTime=" + burstTime +  
 ", state=" + state +  
 (core != null ? (", core number=" + core.getNumber()) : "") +  
 '}' + "\n";  
 }  
}

2. листинг кода класса Queue:

package com.company.classes;  
  
import java.util.\*;

public class Queue {  
 private ArrayList<Process> queue;  
 private int lastID;  
  
 public Queue() {  
 this.queue = new ArrayList<>();  
 this.lastID = 1;  
 }  
  
 public Process get(int i) {  
 return queue.get(i);  
 }  
  
 public int getSize() {  
 return queue.size();  
 }  
  
 public void add(Process process) {  
 this.queue.add(process);  
 }  
  
 public void add() {  
 Process process = new Process(this.lastID++);  
  
 this.add(process);  
 }  
  
 public void add(final int N) {  
 for (int i = 0; i < N; i++) {  
 this.add();  
 }  
 }  
  
 public void remove(int i){  
 this.queue.remove(i);  
 }  
  
 public void remove(Process process){  
 this.queue.remove(process);  
 }  
  
 public synchronized void sortByPriorityAndArrivalTime(State state){  
 Process[] sortedQueue = this.queue.toArray(new Process[0]);  
 int temp;  
 String stemp;  
 for(int i = 0; i < sortedQueue.length; i++){  
 if(sortedQueue[i].getState() != state.*Ready*)  
 continue;  
 for(int j = 0; j <sortedQueue.length - i - 1; j++){  
  
 if(sortedQueue[j].getArrivalTime() > sortedQueue[j + 1].getArrivalTime()){  
 temp = sortedQueue[j].getArrivalTime();  
 sortedQueue[j].setArrivalTime(sortedQueue[j+1].getArrivalTime());  
 sortedQueue[j + 1].setArrivalTime(temp);  
  
 temp = sortedQueue[j].getBurstTime();  
 sortedQueue[j].setBurstTime(sortedQueue[j+1].getBurstTime());  
 sortedQueue[j + 1].setBurstTime(temp);  
  
 temp = sortedQueue[j].getPriority();  
 sortedQueue[j].setPriority(sortedQueue[j+1].getPriority());  
 sortedQueue[j + 1].setPriority(temp);  
  
 stemp = sortedQueue[j].getName();  
 sortedQueue[j].setName(sortedQueue[j+1].getName());  
 sortedQueue[j + 1].setName(stemp);  
 }  
 if(sortedQueue[j].getArrivalTime() == sortedQueue[j+1].getArrivalTime()){  
 if(sortedQueue[j].getPriority() > sortedQueue[j+1].getPriority()){  
 temp = sortedQueue[j].getArrivalTime();  
 sortedQueue[j].setArrivalTime(sortedQueue[j+1].getArrivalTime());  
 sortedQueue[j + 1].setArrivalTime(temp);  
  
 temp = sortedQueue[j].getBurstTime();  
 sortedQueue[j].setBurstTime(sortedQueue[j+1].getBurstTime());  
 sortedQueue[j + 1].setBurstTime(temp);  
  
 temp = sortedQueue[j].getPriority();  
 sortedQueue[j].setPriority(sortedQueue[j+1].getPriority());  
 sortedQueue[j + 1].setPriority(temp);  
  
 stemp = sortedQueue[j].getName();  
 sortedQueue[j].setName(sortedQueue[j+1].getName());  
 sortedQueue[j + 1].setName(stemp);  
 }  
 }  
 }  
 }  
 }  
public int getLastID() {  
 return lastID;  
 }  
  
 public void setLastID(int lastID) {  
 this.lastID = lastID;  
 }  
  
 public int getCountByState(State state){  
 int count = 0;  
 for(int i = 0 ; i < queue.size(); i++){  
 if(queue.get(i).getState() == state)  
 count++;  
 }  
 return count;  
 }  
  
 @Override  
 public String toString() {  
 String result = "";  
 for (Process process : queue) {  
 result += process;  
 }  
 return result;  
 }  
}

3. листинг кода класса Scheduler:

package com.company.classes;  
  
import java.util.Timer;  
import java.util.TimerTask;  
  
public class Scheduler {  
 private Queue jobsQueue;  
 private Queue readyQueue;  
 private Queue waitingQueue;  
 private Queue rejectedQueue;  
  
 private MemoryScheduler memoryScheduler;  
 private TactGenerator tactGenerator;  
  
 private static CPU *cpu*;  
 private static Device *device*;  
  
 public Scheduler(final int cpuCoresNumber, int memoryVolume) {  
 this.jobsQueue = new Queue();  
 this.readyQueue = new Queue();  
 this.rejectedQueue = new Queue();  
 this.waitingQueue = new Queue();  
  
 this.memoryScheduler = new MemoryScheduler();  
 Configuration.*memoryVolume* = memoryVolume;  
  
 *cpu* = new CPU(cpuCoresNumber);  
 *device* = new Device();  
  
 tactGenerator = new TactGenerator();  
 }  
  
 public void addProcess(int number) {  
 jobsQueue.add(number);  
 for (int i = 0; i < jobsQueue.getSize(); i++) {  
 var process = jobsQueue.get(i);  
 if (process.getState() == State.*New*)  
 initProcess(process);  
 }  
  
 jobsQueue.sortByPriorityAndArrivalTime(State.*Ready*);  
 }  
  
public void add(Process process) {  
 jobsQueue.add(process);  
 }  
  
 public void addProcessRandom(int time) {  
 Timer timer = new Timer();  
  
 TimerTask timerTask = new TimerTask() {  
 public void run() {  
 jobsQueue.add(1);  
 jobsQueue.sortByPriorityAndArrivalTime(State.*Ready*);  
 }  
 };  
 timer.schedule(timerTask, time);  
 }  
  
 public void init() {  
  
 Timer timer = new Timer();  
 timer.schedule(tactGenerator, 1000, 1000);  
 tactGenerator.run();  
  
memoryScheduler.add(new MemoryBlock(200));  
 memoryScheduler.add(new MemoryBlock(1200));  
 memoryScheduler.add(new MemoryBlock(1200));  
 memoryScheduler.add(new MemoryBlock(1200));  
}  
  
 public void schedule() {  
  
 do {  
 for (int i = 0; i < readyQueue.getSize(); i++) {  
 var process = readyQueue.get(i);  
  
 if (process.getState() == State.*Ready*) {  
 for (Core core : *cpu*.getCores()) {  
 if (core.isIdle()) {process.setState(State.*Running*);  
 process.setRunTime(TactGenerator.*getTime*());core.setIdle(false);  
 process.setCore(core);  
 break;  
 }  
 }  
 }  
 if (process.getState() != State.*Running*)  
 continue;  
  
 process.setBurstTime(TactGenerator.*getTime*() - process.getRunTime() - process.getIdleTime());  
  
 if (process.getBurstTime() >= process.getTime()) {  
  
 process.setState(State.*Terminated*);  
 System.*out*.println("////////////////////////////////////////////");  
 System.*out*.println(jobsQueue);  
 System.*out*.println("............................................");  
 System.*out*.println(*cpu*);  
 System.*out*.println("............................................");  
 System.*out*.println("--------------------------------------------");  
 System.*out*.println(memoryScheduler);  
 System.*out*.println("--------------------------------------------");  
 System.*out*.println("////////////////////////////////////////////");  
 }  
 }  
  
 for (int i = 0; i < readyQueue.getSize(); i++) {  
 var process = readyQueue.get(i);  
  
 if (process.getState() == State.*Waiting*) {  
 waitingQueue.add(process);  
 readyQueue.remove(process);  
 } else if (process.getState() == State.*Terminated*)  
 readyQueue.remove(process);  
 }  
  
 for (int i = 0; i < waitingQueue.getSize(); i++) {  
 var process = waitingQueue.get(i);  
  
 if (process.getState() == State.*Running*) {  
 readyQueue.add(process);  
 waitingQueue.remove(process);  
 }  
 }  
  
 for (int i = 0; i < jobsQueue.getSize(); i++) {  
 var process = jobsQueue.get(i);  
 if (process.getState() == State.*New*) {  
 initProcess(process);  
 readyQueue.sortByPriorityAndArrivalTime(State.*Ready*);  
}  
 }  
 try {  
 Thread.*sleep*(500);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 } while (true);  
 }  
  
 public MemoryScheduler getMemoryScheduler() {  
 return memoryScheduler;  
 }  
  
 private void initProcess(Process process) {  
 var block = memoryScheduler.fillMemoryBLock(process.getMemory());  
 if (block != null) {  
 process.setState(State.*Ready*);  
 process.setMemoryBlock(block);  
 process.setDevice(*device*);  
 readyQueue.add(process);  
 } else {  
rejectedQueue.add(process);  
 jobsQueue.remove(process);  
 }  
 }  
  
public Queue getJobsQueue() {  
 return jobsQueue;  
 }  
  
 public void setJobsQueue(Queue jobsQueue) {  
 this.jobsQueue = jobsQueue;  
 }  
  
 public Queue getReadyQueue() {  
 return readyQueue;  
 }  
  
 public void setReadyQueue(Queue readyQueue) {  
 this.readyQueue = readyQueue;  
 }  
  
 public Queue getWaitingQueue() {  
 return waitingQueue;  
 }  
  
 public void setWaitingQueue(Queue waitingQueue) {  
 this.waitingQueue = waitingQueue;  
 }  
  
 public Queue getRejectedQueue() {  
 return rejectedQueue;  
 }  
  
 public void setRejectedQueue(Queue rejectedQueue) {  
 this.rejectedQueue = rejectedQueue;  
 }  
  
 public static CPU getCpu() {  
 return *cpu*;  
 }  
  
 public static Device getDevice() {  
 return *device*;  
 }  
  
 @Override  
 public String toString() {  
 return "Scheduler {" + "\n" +  
 "jobs queue: [ " + "\n" + jobsQueue + " ]" + " \n" +  
 "rejected queue: [" + "\n" + rejectedQueue + " ]" + " \n" +  
 "CPU:" + *cpu* + "\n" +  
 "memoryScheduler: " + memoryScheduler +  
 '}';  
 }  
}

4. листинг кода класса Configuration:

package com.company.classes;  
  
public class Configuration {  
 public static int *memoryVolume*;public static final int *maxPriority* = 31;  
}

5. листинг кода класса MemoryBlock:

package com.company.classes;  
  
import java.util.Comparator;

public class MemoryBlock {  
 private int availableMemory;  
  
 public static Comparator<MemoryBlock> *byAvailableMemorySize* = ((o1, o2) -> o2.availableMemory - o1.availableMemory);  
  
 public MemoryBlock(int availableMemory){  
 this.availableMemory = availableMemory;  
 }  
  
 public int getAvailableMemory() {  
 return availableMemory;  
 }  
  
 public void setAvailableMemory(int availableMemory) {  
 this.availableMemory = availableMemory;  
 }  
  
 @Override  
 public String toString() {  
 return "{" + availableMemory + '}';  
 }  
}

6. листинг кода класса MemoryScheduler:

package com.company.classes;  
  
import java.util.ArrayList;  
import java.util.Collections;  
import java.util.Comparator;  
  
public class MemoryScheduler {  
 private static ArrayList<MemoryBlock> *memoryBlocks* = new ArrayList<>();  
 private MemoryBlock biggestBlock;  
  
 public MemoryBlock getBiggestBlock() {  
 return biggestBlock;  
 }  
  
 public void setBiggestBlock(MemoryBlock biggestBlock) {  
 this.biggestBlock = biggestBlock;  
 }  
  
public MemoryBlock fillMemoryBLock(int memorySize) {biggestBlock = Collections.*max*(*memoryBlocks*, Comparator.*comparing*(s -> s.getAvailableMemory()));  
  
 if(memorySize <= biggestBlock.getAvailableMemory()){  
 biggestBlock.setAvailableMemory(biggestBlock.getAvailableMemory() - memorySize);  
 return biggestBlock;  
 }  
 return null;  
 }  
  
public void add(MemoryBlock memoryBlock){  
 *memoryBlocks*.add(memoryBlock);  
 }  
  
 public MemoryBlock get(int i){  
 return *memoryBlocks*.get(i);  
 }  
  
 public int getSize(){  
 return *memoryBlocks*.size();  
 }  
  
 public static String print() {  
 String result = "[";  
 for(MemoryBlock memoryBlock : *memoryBlocks*){  
 result+=memoryBlock+" ";  
 }  
 return result + "]";  
 }  
  
  
  
 @Override  
 public String toString() {  
 String result = "[ ";  
 for (int i = 0; i < *memoryBlocks*.size(); i++){  
 result+=*memoryBlocks*.get(i) + " ";  
 }  
 return result + "]";  
 }  
}

7. листинг кода класса Resource:

package com.company.classes;  
  
public class Resource {  
 protected boolean isIdle = true;  
 protected int resourceNumber;  
  
 public Resource(int number) {  
 this.resourceNumber = number;  
 }  
  
 public int getNumber() {  
 return resourceNumber;  
 }  
  
 public boolean isIdle(){  
 return isIdle;  
 }  
  
 public void setIdle(boolean idle){  
 isIdle = idle;  
 }  
  
 public String getState() {  
 return isIdle ? "Idle" : "Busy";  
 }  
  
 @Override  
 public String toString() {  
 return "resource #" + this.resourceNumber;  
 }  
}

8. листинг кода класса Device:

package com.company.classes;  
  
public class Device {  
  
 private Resource[] resources;  
  
 public Device(){  
 int randomNumber = Utils.*getRandomInteger*(3,5);  
  
 resources = new Resource[randomNumber];  
 for(int i = 0; i < randomNumber; i++){  
 resources[i] = new Resource(i);  
 }  
 }  
  
 public Resource[] getResources() {  
 return resources;  
 }  
  
 public Resource getResource(int resource){  
 return resources[resource];  
 }  
  
 public synchronized int requestResource(){  
 for(Resource resource : resources){  
 if(resource.isIdle() == true){  
 resource.setIdle(false);  
 System.*out*.println("Found idle resource " + resource.getNumber());  
  
 return resource.getNumber();  
 }  
 }  
 return -1;  
 }  
  
 public void doWork(int number){  
 try {  
 int sleepTime = Utils.*getRandomInteger*(10000,20000);  
 System.*out*.println("Do work: " + number + " Sleep time: " + sleepTime);  
 Thread.*sleep*(sleepTime);  
 System.*out*.println("Do work: " + number + " We are out");  
 resources[number].setIdle(true);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
  
 @Override  
 public String toString() {  
 String result = "[ ";  
 for (Resource resource: resources){  
 result += resource + " is "  
 + (resource.isIdle() ? "Idle" : "Busy") + "; ";  
 }  
 return result + "]";  
 }  
}

9. листинг кода класса CPU:

package com.company.classes;  
  
public class CPU {  
  
 private Core[] cores;  
  
 public CPU(final int coresNumber){  
 cores = new Core[coresNumber];  
 for(int i = 0; i < coresNumber; i++){  
 cores[i] = new Core(i);  
 }  
 }  
  
 public Core[] getCores() {  
 return cores;  
 }  
  
 public void setCores(Core[] cores) {  
 this.cores = cores;  
 }  
  
 public Core getCore(int core){  
 return cores[core];  
 }  
  
 @Override  
 public String toString() {  
 String result = "[ ";  
 for (Core core : cores){  
 result += core + " is "  
 + (core.isIdle() ? "Idle" : "Busy") + "; ";  
 }  
 return result + "]";  
 }  
}

10. листинг кода класса CPU:

package com.company.classes;  
  
public class Core extends Resource {  
  
 public Core(int number) {  
 super(number);  
 }  
  
 @Override  
 public String toString() {  
 return "core #" + this.resourceNumber;  
 }  
}

11. листинг кода класса TactGenerator:

package com.company.classes;  
import java.util.TimerTask;  
  
public class TactGenerator extends TimerTask {  
 public static int getTime() {  
 return *time*;  
 }  
  
 public static void incTime(int tact)  
 {  
 *time*+=tact;  
 }  
  
 public static void incTime()  
 {  
 *time*++;  
 }  
  
 private static int *time*;  
  
 @Override  
 public void run() {  
  *incTime*();  
 }  
 }  
}

12. листинг кода класса Utils:

package com.company.classes;  
  
import java.util.Random;  
  
public class Utils {  
 public static Random *random* = new Random();  
 public static int getRandomInteger(int size){  
 return *random*.nextInt(size + 1);  
 }  
  
 public static int getRandomInteger(int left, int right){  
 return left + *random*.nextInt(right - left + 1);  
 }  
}

13. листинг кода класса Controller:

package com.company;  
  
import com.company.classes.\*;  
import com.company.classes.Process;  
import javafx.beans.value.ChangeListener;  
import javafx.beans.value.ObservableValue;  
import javafx.collections.FXCollections;  
import javafx.collections.ObservableList;  
import javafx.fxml.FXML;  
import javafx.scene.control.TableColumn;  
import javafx.scene.control.TableView;  
import javafx.scene.control.TextField;  
import javafx.scene.control.cell.PropertyValueFactory;  
  
import java.util.Timer;  
import java.util.TimerTask;  
  
public class Controller {  
 @FXML  
 private TableView<Process> tableView;  
 @FXML  
 private TableColumn<Process, Integer> processPriorityCol;  
 @FXML  
 private TableColumn<Process, String> processNameCol;  
 @FXML  
 private TableColumn<Process, Integer> processTimeCol;  
 @FXML  
 private TableColumn<Process, Integer> processMemoryCol;  
 @FXML  
 private TableColumn<Process, Integer> processArrivalTimeCol;  
 @FXML  
 private TableColumn<Process, Integer> processBurstTimeCol;  
 @FXML  
 private TableColumn<Process, State> processStateCol;  
 @FXML  
 private TextField textFieldAddProcesses;  
 @FXML  
 private TextField textFieldKillProcess;  
 @FXML  
 private TableView<MemoryBlock> memoryBlockTableView;  
 @FXML  
 private TableColumn<MemoryBlock, Integer> availableMemoryCol;  
 @FXML  
 private TableView<Core> cpuTableView;  
 @FXML  
 private TableColumn<Core, Integer> availableCpuNumberCol;  
 @FXML  
 private TableColumn<Core, Boolean> availableCpuStateCol;  
 @FXML  
 private TableView<Resource> resourceTableView;  
 @FXML  
 private TableColumn<Resource, Integer> availableResourceNumberCol;  
 @FXML  
 private TableColumn<Resource, Boolean> availableResourceStateCol;  
 @FXML  
 private TextField rejectedData;  
 @FXML  
 private TextField totalWaitingDataField;  
 @FXML  
 private TextField totalTerminatedDataField;

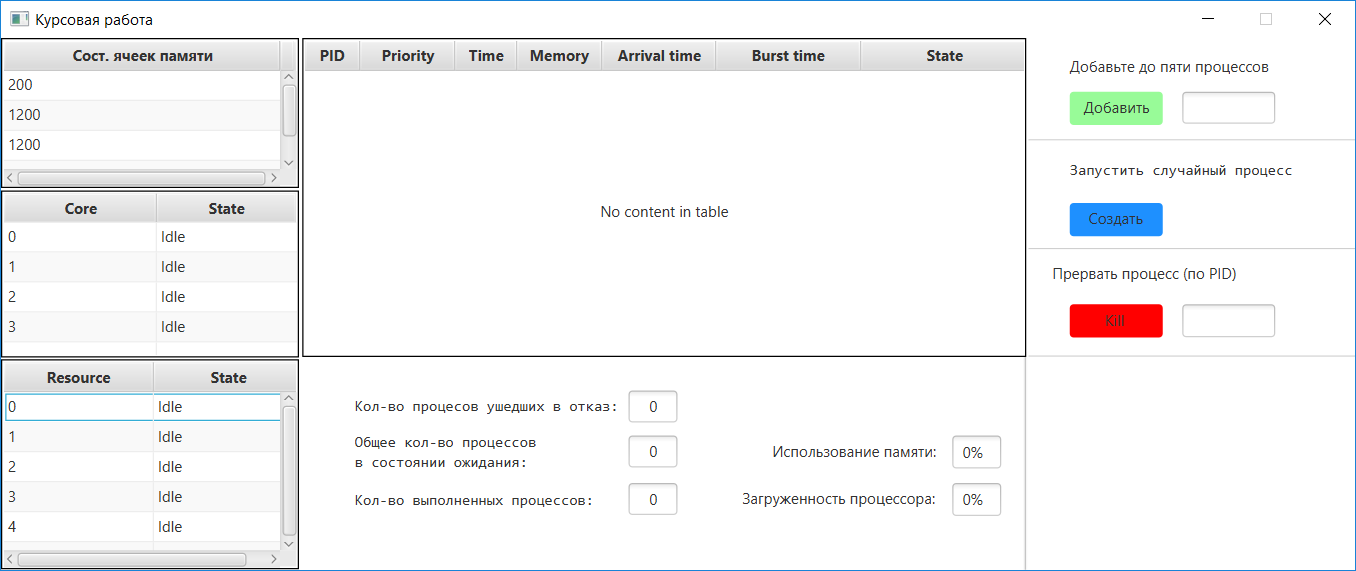
private Main main;  
  
 public Controller() {  
 }  
  
 Scheduler scheduler = new Scheduler(4, 4096);  
 private ObservableList<com.company.classes.Process> processObservableList = FXCollections.*observableArrayList*();  
 private ObservableList<com.company.classes.MemoryBlock> memoryObservableList = FXCollections.*observableArrayList*();  
 private ObservableList<com.company.classes.Core> cpuObservableList = FXCollections.*observableArrayList*();  
 private ObservableList<com.company.classes.Resource> resourceObservableList = FXCollections.*observableArrayList*();  
  
 @FXML  
 private void initialize() {  
 tableView.setColumnResizePolicy(TableView.*CONSTRAINED\_RESIZE\_POLICY*);  
 memoryBlockTableView.setColumnResizePolicy(TableView.*CONSTRAINED\_RESIZE\_POLICY*);  
 resourceTableView.setColumnResizePolicy(TableView.*CONSTRAINED\_RESIZE\_POLICY*);  
 cpuTableView.setColumnResizePolicy(TableView.*CONSTRAINED\_RESIZE\_POLICY*);  
 processPriorityCol.setCellValueFactory(new PropertyValueFactory<>("priority"));  
 processNameCol.setCellValueFactory(new PropertyValueFactory<>("id"));  
 processTimeCol.setCellValueFactory(new PropertyValueFactory<>("time"));  
 processMemoryCol.setCellValueFactory(new PropertyValueFactory<>("memory"));  
 processArrivalTimeCol.setCellValueFactory(new PropertyValueFactory<>("arrivalTime"));  
 processBurstTimeCol.setCellValueFactory(new PropertyValueFactory<>("burstTime"));  
 processStateCol.setCellValueFactory(new PropertyValueFactory<>("state"));  
 availableMemoryCol.setCellValueFactory(new PropertyValueFactory<>("availableMemory"));  
  
 availableCpuStateCol.setCellValueFactory(new PropertyValueFactory<>("state"));  
 availableCpuNumberCol.setCellValueFactory(new PropertyValueFactory<>("number"));  
  
 availableResourceNumberCol.setCellValueFactory(new PropertyValueFactory<>("number"));  
 availableResourceStateCol.setCellValueFactory(new PropertyValueFactory<>("state"));  
  
 scheduler.init();  
  
 Thread schedulerThread = new Thread(() -> {  
 scheduler.schedule();  
 });  
 schedulerThread.start();  
  
 Timer timer = new Timer();  
 TimerTask timerTask = new TimerTask() {  
  
 int totalReady = 0;  
 int totalWaiting = 0;  
 int totalTerminated = 0;  
  
 public void run() {  
tableView.getItems().clear();  
 for (int i = 0; i < scheduler.getJobsQueue().getSize(); i++) {  
 processObservableList.add(scheduler.getJobsQueue().get(i));  
 }  
 tableView.setItems(processObservableList);  
cpuTableView.getItems().clear();  
 var cpu = Scheduler.*getCpu*();  
 if(cpu != null) {  
 for (int i = 0; i < cpu.getCores().length; i++) {  
 cpuObservableList.add(cpu.getCore(i));  
 }  
 }  
 cpuTableView.setItems(cpuObservableList);  
resourceTableView.getItems().clear();  
 var device = Scheduler.*getDevice*();  
 if(device != null) {  
 for (int i = 0; i < device.getResources().length; i++) {  
 resourceObservableList.add(device.getResource(i));  
 }  
 }  
 resourceTableView.setItems(resourceObservableList);  
memoryBlockTableView.getItems().clear();  
 MemoryScheduler memoryScheduler = scheduler.getMemoryScheduler();  
 for (int i = 0; i < memoryScheduler.getSize(); i++) {  
 memoryObservableList.add(memoryScheduler.get(i));  
 }  
 memoryBlockTableView.setItems(memoryObservableList);  
  
 int rejectedCount = scheduler.getRejectedQueue().getSize();  
 rejectedData.setText(String.*valueOf*(rejectedCount));  
  
 int readyCount = scheduler.getReadyQueue().getCountByState(State.*Ready*);  
 int runningCount = scheduler.getReadyQueue().getCountByState(State.*Running*);  
 readyCount += runningCount;  
 totalReady += readyCount;int waitingCount = scheduler.getWaitingQueue().getCountByState(State.*Waiting*);  
 totalWaiting += waitingCount;  
 totalWaitingDataField.setText(String.*valueOf*(totalWaiting));  
  
 int terminatedCount = scheduler.getJobsQueue().getCountByState(State.*Terminated*);  
 totalTerminated += terminatedCount;  
 totalTerminatedDataField.setText(String.*valueOf*(totalTerminated));  
  
 }  
 };  
 timer.schedule(timerTask, 1000, 1000);  
}  
  
 @FXML  
 private void addNumberOfProcessesButton() {  
 try {  
 textFieldAddProcesses.focusedProperty().addListener(new ChangeListener<Boolean>() {  
 @Override  
 public void changed(ObservableValue<? extends Boolean> observableValue, Boolean aBoolean, Boolean t1) {  
 if (t1) {  
 textFieldAddProcesses.setStyle("-fx-control-inner-background: #ffffff");  
 }  
 }  
 });  
 Integer number = Integer.*parseInt*(textFieldAddProcesses.getText());  
 if (number > 0 && number <= 5) {  
 scheduler.addProcess(number);  
 textFieldAddProcesses.clear();  
 } else {  
 textFieldAddProcesses.setStyle("-fx-control-inner-background: #ff3d3b");  
 }  
} catch (NumberFormatException exception) {  
 textFieldAddProcesses.clear();  
 }  
 }  
  
 @FXML  
 private void addNumberOfProcessesInRandomTimeButton() {  
 int procNumber = Utils.*getRandomInteger*(1, 1);  
 int delay = 0;  
 for (int i = 0; i < procNumber; i++) {  
 delay = Utils.*getRandomInteger*(10000, 20000);  
 scheduler.addProcessRandom(delay);  
 }  
 }  
  
 @FXML  
 private void killProcess() {  
 textFieldKillProcess.focusedProperty().addListener(new ChangeListener<Boolean>() {  
 @Override  
 public void changed(ObservableValue<? extends Boolean> observableValue, Boolean aBoolean, Boolean t1) {  
 if (t1) {  
 textFieldKillProcess.setStyle("-fx-control-inner-background: #ffffff");  
 }  
 }  
 });  
  
 int killed = -1;  
 try {  
 killed = Integer.*parseInt*(textFieldKillProcess.getText());  
 } catch (NumberFormatException e){  
  
 return;  
 }  
  
 for (int i = 0; i < scheduler.getJobsQueue().getSize(); i++) {  
 if (scheduler.getJobsQueue().get(i).getId() == killed) {  
 scheduler.getJobsQueue().remove(i);  
 textFieldKillProcess.clear();  
 textFieldKillProcess.setStyle("-fx-control-inner-background: #ffffff");  
 } else {  
 textFieldKillProcess.clear();  
 }  
 }  
 for (int i = 0; i < scheduler.getReadyQueue().getSize(); i++) {  
 if (scheduler.getReadyQueue().get(i).getId() == killed) {  
 scheduler.getReadyQueue().remove(i);  
 textFieldKillProcess.clear();  
 textFieldKillProcess.setStyle("-fx-control-inner-background: #ffffff");  
 } else {  
 textFieldKillProcess.clear();  
 }  
 }  
 for (int i = 0; i < scheduler.getWaitingQueue().getSize(); i++) {  
 var waitingProcess = scheduler.getWaitingQueue().get(i);  
 if (waitingProcess.getId() == killed) {  
 var resourceId = waitingProcess.getResourceId();  
 Scheduler.*getDevice*().getResource(resourceId).setIdle(true);  
 scheduler.getWaitingQueue().remove(i);  
 textFieldKillProcess.clear();  
 textFieldKillProcess.setStyle("-fx-control-inner-background: #ffffff");  
 } else {  
 textFieldKillProcess.clear();  
 }  
 }  
 }  
}

14. листинг кода класса Main:

import javafx.application.Application;  
import javafx.application.Platform;  
import javafx.fxml.FXMLLoader;  
import javafx.scene.Parent;  
import javafx.scene.Scene;  
import javafx.stage.Stage;  
  
public class Main extends Application {  
  
 @Override  
 public void start(Stage primaryStage) throws Exception{  
 Parent root = FXMLLoader.*load*(getClass().getResource("viewFile1.fxml"));  
 primaryStage.setTitle("Hello World");  
 primaryStage.setScene(new Scene(root, 959, 426));  
 primaryStage.setResizable(false);  
 primaryStage.setOnCloseRequest(e -> {  
 Platform.*exit*();  
 System.*exit*(0);  
 });  
 primaryStage.show();  
 }  
  
  
 public static void main(String[] args) {  
 *launch*(args);  
 }  
  
}

**ИНСТРУКЦИЯ**

Для начала роботы, добавьте процесс. Сделать можно это либо задав значение (до пяти) в текстовом поле возле зеленой кнопки «Добавить», после чего нажать её. Или же вы можете инициировать создание процесса через случайный промежуток времени, нажав на синюю кнопку «Создать». При желании, вы можете уничтожить процесс, введя его PID в текстовое поле, которое находится возле красной кнопки «Kill», после чего нажать её.



Остальная часть интерфейса ответственна за демонстрацию данных.

# **СПИСОК ЛИТЕРАТУРЫ**

1. А. Ю. Молчанов Системное программное обеспечение: /. - СПб.; М.; Нижний Новгород: Питер: Питер принт, 2003. - 395 с.

2. Э. Таненбаум Современные операционные системы. 2-е изд. - СПб.: Питер, 2007. - 1038 с.

3. Х. М. Дейтел, П. Д. Дейтел, Д. Р. Чофнес Операционные системы: [в 2 т.] : пер. с англ./; под ред. С. М. Молявко. - М.: Бином-Пресс, 2006 - 1023 с.

4. Н. А. Прохоренок Java FX – СПб.: БХВ\_Петербург, 2020 – 768с.: ил. – (в подлиннике)

5. Шилдт, Герберт. Java. Полное руководство, 10-е изд.: Пер. с англ. -СПб.ООО "Альфакнига'; 2018. - 1488 с.: ил. - Парал. тит. англ.

6. Прохоренок Н. А.П84 Основы Java. — СПб.: БХВ-Петербург, 2017. — 704 с.: ил.

7. Лав Р. Л13 Linux. Системное программирование. 2-е изд. — СПб.: Питер, 2014. — 448 с.: ил. — (Серия «Бестселлеры O’Reilly»).