



Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут ім. Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

ЛАБОРАТОРНА РОБОТА №3
З ДИСЦИПЛІНИ “ОРГАНІЗАЦІЯ ОБЧИСЛЮВАЛЬНИХ
ПРОЦЕСІВ”

Виконав:

Студент III курсу ФІОТ
групи ІО-82
Шендріков Євгеній

Перевірив:

Сімоненко В. П.

Завдання

1. Реалізувати запропонований алгоритм планування для однопроцесорної системи.

2. Реалізувати імітацію появи процесів в системі випадковим чином. Для кожного процесу генерується час його появи в системі, кількість тактів необхідні для роботи процесу (який вимірюється в тактах з початке моделювання) і якщо необхідно його пріоритет.

3. Параметрами системи (задаються користувачем) є діапазон часів роботи процесів, інтенсивність вхідного потоку, квант, діапазон пріоритетів (для пріоритетних дисциплін обслуговування). В систему може надходити будь-яка кількість процесів, система не має бути перевантажена. Закінчення дослідження визначається користувачем кількістю тактів моделювання.

4. Для кожного процесу з часом виконання t обчислювати:

- T – загальний час перебування процесу в системі.
- Втрачений час $M = T - t$;
- Коефіцієнт реактивності $R = t / T$;
- Штрафне відношення $P = T / t$;

5. Побудувати графіки середніх значень R та P в залежності від інтенсивності при використанні алгоритму планування процесів. Програма може генерувати значення для графіків, а самі графіки можуть будуватися сторонньою програмою.

6. Для порівняння реалізувати також простий алгоритм планування, вказаний в завданні. Для простого алгоритму планування побудувати такі ж графіки, що задані в пункті 5.

7. Побудувати графіки: залежності середнього часу очікування від інтенсивності вхідного потоку процесів, проценту простою процесору від інтенсивності вхідного потоку, залежності кількості процесів від часу очікування при фіксованій інтенсивності вхідного потоку процесів.

8. Пояснити форму графіків.

9. При реалізації дисципліни обслуговування намагатись досягти константної складності роботи з чергою.

Варіант

$$27 \% 19 + 1 = 9 \text{ (PSJF та RR)}$$

Опис роботи дисциплін обслуговування

Round Robin

Алгоритм планування **Round Robin** (RR) в основному розроблено для систем з розподілом часу. Цей алгоритм аналогічний FCFS, але при плануванні RR додається переривання, яке дозволяє системі перемикатися між процесами.

Кожному процесу відводиться фіксований час, званий квантом, для виконання. Як тільки процес виконується протягом заданого періоду часу, цей процес переривається, і інший процес виконується протягом заданого періоду часу.

Перемикання контексту використовується для збереження станів витіснених процесів. Цей алгоритм простий і легкий в реалізації, і найголовніше - цей алгоритм не вимагає «голодування», оскільки всі процеси отримують значну частку ресурсів ЦП. Тут важливо зазначити, що тривалість кванта часу зазвичай становить від 10 до 100 мілісекунд.

Деякі важливі характеристики алгоритму циклічного перебору (RR) полягають в наступному:

1. Алгоритм циклічного планування відноситься до категорії алгоритмів з витісненням.
2. Цей алгоритм - один з найстаріших, найпростіших і справедливіших алгоритмів.
3. Цей алгоритм є алгоритмом реального часу, оскільки він реагує на подію протягом певного періоду часу.
4. У цьому алгоритмі часовий інтервал повинен бути мінімальним, який призначається конкретному завданню, яке необхідно обробити. Хоча це може відрізнятися для різних операційних систем.

Зі зменшенням значення кванту часу:

- Збільшується кількість перемикань контексту.
- Час відгуку зменшується
- Шанси «голоду» в цьому випадку зменшуються.

При меншому значенні часового кванта він стає кращим з точки зору часу відгуку.

Зі збільшенням значення кванту часу:

- Кількість перемикань контексту зменшується.
- Час відгуку збільшується.
- Шанси «голоду» в цьому випадку зростають.

Для вищого значення часового кванта він стає кращим з точки зору кількості перемикань контексту.

Якщо значення кванту часу зростає, то планування Round Robin, як правило, стає плануванням FCFS. У цьому випадку, коли значення кванту часу прагне до нескінченності, то планування Round Robin стає плануванням FCFS.

Таким чином, ефективність планування Round Robin в основному залежить від значення кванту часу. І значення часового кванта має бути таким, щоб воно не було ні занадто великим, ні занадто малим.

Preemptive SJF

Preemptive Shortest Job First (PSJF) відомий також як **Shortest Remaining Time First (SRTF)**.

За допомогою цього алгоритму першим для виконання вибирається процес, який має найменшу кількість часу, що залишився до завершення.

Отже, в основному в PSJF процеси плануються відповідно до найкоротшого часу, що залишився.

Однак алгоритм PSJF передбачає більше накладних витрат, ніж планування SJF, оскільки в PSJF ОС часто потрібна для того, щоб контролювати час процесора завдань у черзі готовності (ready queue) і виконувати перемикання контексту.

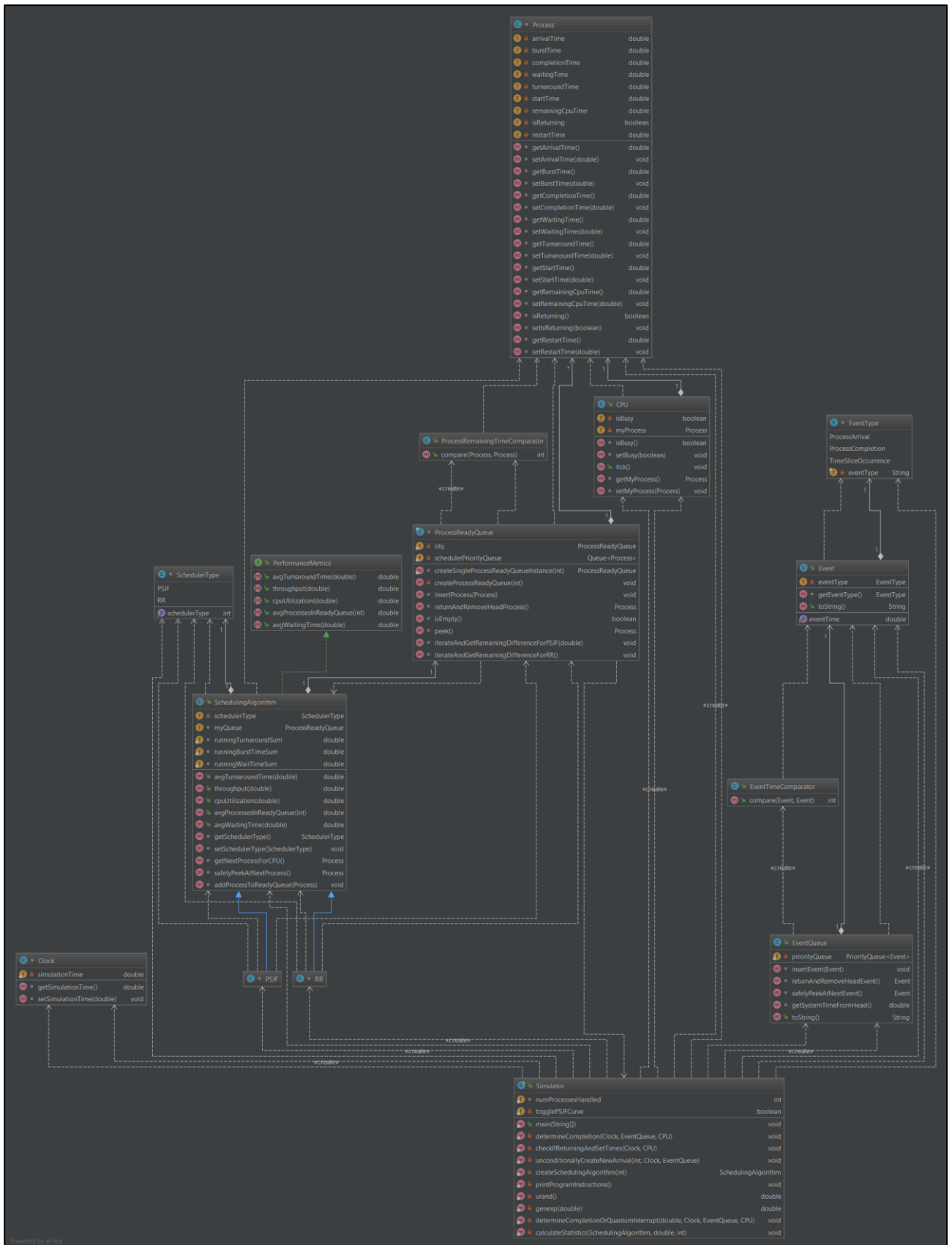
В алгоритмі планування PSJF виконання будь-якого процесу може бути зупинено через певний проміжок часу. Після прибуття кожного процесу, короткостроковий планувальник планує ці процеси зі списку доступних процесів та запущених процесів, які мають найменший залишковий час.

Після того, як всі процеси будуть доступні в черзі готовності, витіснення (preemption) не буде виконано, і тоді алгоритм буде працювати так само, як планування SJF.

Лістинг та опис програми

Як буде видно з діаграми класів на наступній сторінці мною був розроблений достатньо об'ємний проект, щоб покрити вимоги завдання. Далі також буде наведений короткий опис до кожного створеного класу та його загальне призначення.

Діаграма класів



Simulator.java

Головний клас програми. Цей клас керує моделюванням одного екземпляра планувальника за допомогою аргументів командного рядка, які зазначив користувач. Клас містить основний цикл `while`, який продовжує обробку подій до завершення 10 000 процесів.

При цьому він не зупиняє і не перешкоджає генеруванню нових надходжень процесів, оскільки це необхідно для точного моделювання з достовірними статистичними результатами.

Я експериментував із генерацією всіх 10 000 процесів заздалегідь, але це спричинило проблеми у розподілі та обчислювало значення статистики. Генерування нових надходжень по ходу - кращий підхід, який дає більш точні результати.

У нас є три основних типи подій - *ProcessArrival*, *ProcessCompletion* і *TimeSliceOccurrence*. Останній використовується тільки планувальником Round Robin. Я вирішив не робити четверту, окрему подію для витіснення в PSJF. Оскільки це відбувається під час моделювання, коли це виявляється, я обробляю його тут же.

Кожен раз, коли ми плануємо процес на ЦП, ми перевіряємо, чи є він новим чи тим, що повернувся (вже запущеним раніше). Потім ми можемо перевірити, чи завершиться процес або він вимагає особливої обробки - витіснення для PSJF або переривання для RR.

Якщо подія:

- 1) прибула: створюємо процес і додаємо його до черги планувальника
- 2) завершилась: оновлюємо проміжні значення, необхідні для статистики, щоб планувальник почав виконувати наступний процес у ReadyQueue, якщо він є, і запланувати подію на завершення в майбутньому.

Як вже вказувалось раніше, програма приймає аргументи з командного рядка, загальна структура виглядає наступним чином:

```
java -jar Shendrikov_OOP_Lab3.jar <scheduler_type> <lambda>  
<average_service_time> <quantum>
```

[*scheduler_type*]: значення може бути в діапазоні [1,2].

1 - Планувальник Preemptive Shortest Job First (PSJF)

2 - Планувальник Round Robin (RR) - потрібен 4-й аргумент, що визначає квант.

[*lambda*]: значення інтенсивності, необхідне для пуассонівського процесу, щоб гарантувати експоненціальний час між надходженнями.

[average_service_time]: час обслуговування вибирається відповідно до експоненціального розподілу із середнім часом обслуговування, взятого з цього аргументу (значення в секундах).

[quantum]: необов'язковий аргумент потрібно тільки для Round Robin (scheduler_type = 2) (значення в секундах).

[togglePSJFCurveCurve]: приймає False або True. Необов'язковий аргумент для перемикання кривої PSJF з плоскої (0) на неплоску (1).

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Objects;
import java.util.Random;

import static java.lang.Math.log;

/**
 * @author Jack Shendrikov
 */

public class Simulator {

    static int numProcessesHandled = 0;
    private static boolean togglePSJFCurve = false;

    public static void main(String[] args) throws IOException {

        if (args.length == 0 || args[0].toLowerCase().equals("help") ||
args.length < 4) {
            printProgramInstructions();
        } else {

            /*
             If user provides an optional 5th parameter, we can toggle the
             shape of certain PSJF curves for a different
             interpretation as needed.
            */
            if (args.length == 5) {
                if (args[4].equals("true") || args[4].equals("false")) {
                    togglePSJFCurve = Boolean.parseBoolean(args[4]);
                    System.out.println(togglePSJFCurve);
                }
            }

            // initialize system state variables
            final int algorithmType = Integer.parseInt(args[0]);
            final int lambda = Integer.parseInt(args[1]); // average rate of
arrival

            final double avgServiceTime = Double.parseDouble(args[2]);
            final double quantumForRR = Double.parseDouble(args[3]);

            // initialize simulation clock to 0
            Clock simulationClock = new Clock();
            simulationClock.setSimulationTime(0f);

            EventQueue eventQueue = new EventQueue();
```

```

        Event initialEvent = new Event(EventType.ProcessArrival, 0);
        eventQueue.insertEvent(initialEvent);

        // create the scheduling algorithm and the CPU to handle processes
        SchedulingAlgorithm schedulingAlgorithm =
createSchedulingAlgorithm(algorithmType);
        CPU simulationCPU = new CPU();

        // while we have not processed N Processes to completion,
        // keep going and handle events in the `EventQueue` as needed
        while (numProcessesHandled < 10000) {
            // Set `Clock` to EventTime

simulationClock.setSimulationTime(eventQueue.getSystemTimeFromHead());

            // Do/process next event and remove from `EventQueue`
            Event eventToProcess = eventQueue.returnAndRemoveHeadEvent();
            EventType eventToProcessType = eventToProcess.getEventType();

            /* If event is:
            *   1) an arrival: create a process and add it to the
scheduler's queue
            *   2) a completion: update the intermediate numbers needed for
statistics have scheduler start
            *       executing next process in ReadyQueue if available and
schedule completion event in the future if
            *       RR because we know the completion times. RR is start time
+ quantum.
            */
            if (eventToProcessType == EventType.ProcessArrival) {
                // routine to unconditionally create new arrival event
                unconditionallyCreateNewArrival(lambda, simulationClock,
eventQueue);

                // create the "arriving" process
                Process p = new Process();
                p.setArrivalTime(simulationClock.getSimulationTime()); //
processArrivalTime = eventTime
                p.setBurstTime(genexp(1/avgServiceTime));
                p.setRemainingCpuTime(p.getBurstTime());

                // add new process to scheduler's ready queue
unconditionally
                // only always use a process from the queue, not p directly
Objects.requireNonNull(schedulingAlgorithm).addProcessToReadyQueue(p);

                if (algorithmType == SchedulerType.PSJF.getSchedulerType())
{
                    // CPU not busy, give it a process from queue, no
preemption possible in this case but may have completion
                    if (!simulationCPU.isBusy()) {
simulationCPU.setMyProcess(schedulingAlgorithm.getNextProcessForCPU());
                        simulationCPU.setBusy(true);

                        checkIfReturningAndSetTimes(simulationClock,
simulationCPU);

                        if
(eventQueue.safelyPeekAtNextEvent().getEventType() == EventType.ProcessArrival)
{
                            if ((simulationClock.getSimulationTime() +
simulationCPU.getMyProcess().getRemainingCpuTime())
                                <=
eventQueue.safelyPeekAtNextEvent().getEventTime()) {

```



```

        Event knownCompletion = new
Event(EventType.ProcessCompletion,

simulationCPU.getMyProcess().getRestartTime() +
simulationCPU.getMyProcess().getRemainingCpuTime());
        eventQueue.insertEvent(knownCompletion);
    }
}
} // end CPU IDLE

//else CPU is busy and we may have to preempt if
conditions are met
    else {
        // process ready queue sorted by remTime, not
arrival, so we are not guaranteed sequential processes
        // so, check system time for current time instead
        double elapsedTime =
simulationClock.getSimulationTime() -
simulationCPU.getMyProcess().getRestartTime();
        double oldRemTime =
simulationCPU.getMyProcess().getRemainingCpuTime();
        double newRemTime = oldRemTime - elapsedTime;

        if (newRemTime <= 0) {
            Event knownCompletion = new
Event(EventType.ProcessCompletion,
simulationClock.getSimulationTime() +
oldRemTime);
            eventQueue.insertEvent(knownCompletion);
        }
        else if
(schedulingAlgorithm.safelyPeekAtNextProcess().getRemainingCpuTime() >=
newRemTime) {
simulationCPU.getMyProcess().setRemainingCpuTime(newRemTime);
            determineCompletion(simulationClock, eventQueue,
simulationCPU);
        }

        // else head process has a shorter remTime and we
need to PREEMPT
        // no special event type because preemption happens
at the current system time
        else if
(schedulingAlgorithm.safelyPeekAtNextProcess().getRemainingCpuTime() <
newRemTime) {
simulationCPU.getMyProcess().setRemainingCpuTime(newRemTime);
            Process tempProcess =
simulationCPU.getMyProcess();
simulationCPU.setMyProcess(schedulingAlgorithm.getNextProcessForCPU());
            checkIfReturningAndSetTimes(simulationClock,
simulationCPU);

schedulingAlgorithm.addProcessToReadyQueue(tempProcess);

            //determine completion
            determineCompletion(simulationClock, eventQueue,
simulationCPU);
        }
    } // end CPU busy
} // end PSJF arrival handling

else if (algorithmType ==

```

```

SchedulerType.RR.getSchedulerType()) {
    if (simulationCPU.isBusy()) {} else {

simulationCPU.setMyProcess(schedulingAlgorithm.getNextProcessForCPU());
        simulationCPU.setBusy(true);
        checkIfReturningAndSetTimes(simulationClock,
simulationCPU);
            determineCompletionOrQuantumInterrupt(quantumForRR,
simulationClock, eventQueue, simulationCPU);
        } // end else CPU is IDLE
    } // end RR arrival handling
} // end if to handle Process Arrivals

    else if (eventToProcessType == EventType.ProcessCompletion) {
        /* When an event completes, set its remainingCpuTime to zero
        * increment numProcessesHandled counter.
        * Also the CPU is free to work on another process, so we
must give it one
        */
        numProcessesHandled++;

        if (numProcessesHandled == 10000) {
            System.out.println("10000th process completing now");
        }

        if (algorithmType == SchedulerType.PSJF.getSchedulerType())
{
            simulationCPU.getMyProcess().setRemainingCpuTime(0); //
process is done

simulationCPU.getMyProcess().setCompletionTime(simulationClock.getSimulationTime
());

simulationCPU.getMyProcess().setTurnaroundTime(simulationCPU.getMyProcess().getC
ompletionTime()
-
simulationCPU.getMyProcess().getArrivalTime());
            double completionMinusStart =
simulationCPU.getMyProcess().getCompletionTime() -
simulationCPU.getMyProcess().getStartTime();
            simulationCPU.getMyProcess().setWaitingTime(
                (simulationCPU.getMyProcess().getStartTime() -
simulationCPU.getMyProcess().getArrivalTime()) +
                (completionMinusStart -
simulationCPU.getMyProcess().getBurstTime()));

            // now that a process is complete, update runningSums
that we will use to calculate statistics
            SchedulingAlgorithm.runningBurstTimeSum +=
simulationCPU.getMyProcess().getBurstTime();
            SchedulingAlgorithm.runningTurnaroundSum +=
simulationCPU.getMyProcess().getTurnaroundTime();
            SchedulingAlgorithm.runningWaitTimeSum +=
simulationCPU.getMyProcess().getWaitingTime();

            simulationCPU.setBusy(false);
            if
(!Objects.requireNonNull(schedulingAlgorithm).myQueue.isEmpty()) {
simulationCPU.setMyProcess(schedulingAlgorithm.getNextProcessForCPU());
                simulationCPU.setBusy(true);
                checkIfReturningAndSetTimes(simulationClock,
simulationCPU);

                //determine completion
                Event nextEvent =

```

```

eventQueue.safelyPeekAtNextEvent();
    if (nextEvent.getEventType() ==
EventType.ProcessArrival) {
        double nextArrival = nextEvent.getEventTime();
        double elapsedTime = nextArrival -
simulationCPU.getMyProcess().getRestartTime();
        double oldRemTime =
simulationCPU.getMyProcess().getRemainingCpuTime();
        double newRemTime = oldRemTime - elapsedTime;

        if (newRemTime <= 0) {
            Event knownCompletion = new
Event(EventType.ProcessCompletion,
simulationCPU.getMyProcess().getRestartTime() + oldRemTime);
            eventQueue.insertEvent(knownCompletion);
        } else {
            // we need to preempt when the new process
arrives, not right now
        }
    }
    } else {
        continue;
    }
    // set start time for a new, non-returning process
} // end PSJF completion

else if (algorithmType ==
SchedulerType.RR.getSchedulerType()) {
    simulationCPU.getMyProcess().setRemainingCpuTime(0); //
process is done

simulationCPU.getMyProcess().setCompletionTime(simulationClock.getSimulationTime
());

simulationCPU.getMyProcess().setTurnaroundTime(simulationCPU.getMyProcess().getC
ompletionTime()
-
simulationCPU.getMyProcess().getArrivalTime());
    double completionMinusStart =
simulationCPU.getMyProcess().getCompletionTime() -
simulationCPU.getMyProcess().getStartTime();

//simulationCPU.getMyProcess().setWaitingTime(simulationCPU.getMyProcess().getTu
rnaroundTime()
//
-
simulationCPU.getMyProcess().getBurstTime());
    simulationCPU.getMyProcess().setWaitingTime(
(simulationCPU.getMyProcess().getStartTime() -
simulationCPU.getMyProcess().getArrivalTime())
+ (completionMinusStart -
simulationCPU.getMyProcess().getBurstTime()));

    // now that a process is complete, update runningSums
that we will use to calculate statistics
    SchedulingAlgorithm.runningBurstTimeSum +=
simulationCPU.getMyProcess().getBurstTime();
    SchedulingAlgorithm.runningTurnaroundSum +=
simulationCPU.getMyProcess().getTurnaroundTime();
    SchedulingAlgorithm.runningWaitTimeSum +=
simulationCPU.getMyProcess().getWaitingTime();

    simulationCPU.setBusy(false);

    if
(!Objects.requireNonNull(schedulingAlgorithm).myQueue.isEmpty()) {

```

```

simulationCPU.setMyProcess(schedulingAlgorithm.getNextProcessForCPU());
        simulationCPU.setBusy(true);
        // set start time for a new, non-returning process
        if(!simulationCPU.getMyProcess().isReturning()) {
simulationCPU.getMyProcess().setStartTime(simulationClock.getSimulationTime());
simulationCPU.getMyProcess().setIsReturning(true);
        }

        determineCompletionOrQuantumInterrupt(quantumForRR,
simulationClock, eventQueue, simulationCPU);
        } else {
            continue;
        }
    } // end RR completion
} // end else-if to handle Process Completions
else if (eventToProcessType == EventType.TimeSliceOccurrence) {
simulationCPU.getMyProcess().setRemainingCpuTime(simulationCPU.getMyProcess().getRemainingCpuTime() - quantumForRR);

Objects.requireNonNull(schedulingAlgorithm).myQueue.insertProcess(simulationCPU.getMyProcess());

simulationCPU.setMyProcess(schedulingAlgorithm.getNextProcessForCPU());
        checkIfReturningAndSetTimes(simulationClock, simulationCPU);
        determineCompletionOrQuantumInterrupt(quantumForRR,
simulationClock, eventQueue, simulationCPU);
        } // end time slice occurrence
    } // end while

    if (Objects.requireNonNull(schedulingAlgorithm).getSchedulerType()
== SchedulerType.PSJF && togglePSJFCurve) {
schedulingAlgorithm.myQueue.iterateAndGetRemainingDifferenceForPSJF(simulationClock.getSimulationTime());
    }

    if (schedulingAlgorithm.getSchedulerType() == SchedulerType.RR) {
schedulingAlgorithm.myQueue.iterateAndGetRemainingDifferenceForRR();
    }

    System.out.println("Total sim time: " +
simulationClock.getSimulationTime());
    calculateStatistics(schedulingAlgorithm,
simulationClock.getSimulationTime(), lambda);
    } // end if-else args.length validation
} // end main

/**
 * Used by PSJF algorithm to determine if and when a given process will
 * complete.
 */
private static void determineCompletion(Clock simulationClock, EventQueue
eventQueue, CPU simulationCPU) {
    // determine completion
    Event nextEvent = eventQueue.safelyPeekAtNextEvent();
    if (nextEvent.getEventType() == EventType.ProcessArrival) {
        double nextArrival = nextEvent.getEventTime();
        double _elapsedTime = nextArrival -
simulationClock.getSimulationTime();
        double _oldRemTime =
simulationCPU.getMyProcess().getRemainingCpuTime();

```

```

        double _newRemTime = _oldRemTime - _elapsedTime;

        if (_newRemTime <= 0) {
            Event knownCompletion = new Event(EventType.ProcessCompletion,
                simulationClock.getSimulationTime() + _oldRemTime);
            eventQueue.insertEvent(knownCompletion);
        }
    } // end determineCompletion

    /**
     * Used by multiple schedulers as a generic check to determine if a process
     * is new or returning and set
     * certain parameters accordingly. If a process is new, we set the start
     * time, otherwise we do not so we do
     * not override it.
     */
    private static void checkIfReturningAndSetTimes(Clock simulationClock, CPU
simulationCPU) {
        if (!simulationCPU.getMyProcess().isReturning()) {
simulationCPU.getMyProcess().setStartTime(simulationClock.getSimulationTime());

simulationCPU.getMyProcess().setRestartTime(simulationCPU.getMyProcess().getStar
tTime());
            simulationCPU.getMyProcess().setIsReturning(true);
        } else {
simulationCPU.getMyProcess().setRestartTime(simulationClock.getSimulationTime())
;
        }
    }

    /**
     * This method generates a new arrival event and places it in the event
     * queue.
     */
    private static void unconditionallyCreateNewArrival(int lambda, Clock
simulationClock, EventQueue eventQueue) {
        // routine to unconditionally create new arrival event
        Event newArrival = new Event(EventType.ProcessArrival,
            simulationClock.getSimulationTime() + genexp(lambda));
        eventQueue.insertEvent(newArrival);
    }

    private static SchedulingAlgorithm createSchedulingAlgorithm(int
algorithmType) {
        SchedulingAlgorithm schedulingAlgorithm; // validate that algorithmType
is in range (1,2)
        if (algorithmType > 0 && algorithmType < 3) {

            // create scheduler based on user defined type
            // the scheduler will internally set its type and create its
specific Process Ready Queue
            if (algorithmType == SchedulerType.PSJF.getSchedulerType()) { //
PSJF
                schedulingAlgorithm = new PSJF();
            }
            else { // RR
                schedulingAlgorithm = new RR();
            }
        } else {
            System.out.print("Please enter a valid value for the algorithm type,
in range [1,2].");
            return null;

```

```

    }

    return schedulingAlgorithm;
}

/**
 * This method prints usage instructions to the command line if the user
 * does not specify any
 * command line arguments or types the word 'help'
 */
private static void printProgramInstructions() {
    System.out.println("Event Simulator for 2 scheduling algorithms. ");
    System.out.println("Author: Jack Shendrikov");
    System.out.println("Parameters 1 - 4 are required, including quantum
even if it is not used by the scheduler.");
    System.out.println("java -jar DiscreteEventSimulator.jar
<scheduler_type> <lambda> <avg. svc time> <quantum> <togglePSJFCurve>");
    System.out.println("[scheduler_type] : value can be in the range
[1,2].");
    System.out.println("\t1 - Preemptive Shortest Job First (PSJF)
Scheduler");
    System.out.println("\t2 - Round Robin (RR) Scheduler - requires 4th
argument defining a quantum value.");
    System.out.println("[lambda] : average rate lambda that follows a
Poisson process, to ensure exponential inter-arrival times.");
    System.out.println("[avg. svc time] : the service time is chosen
according to an exponential distribution with an average service time of this
third argument");
    System.out.println("[quantum] : optional argument only required for
Round Robin (scheduler_type = 2). Defines the length of the quantum time
slice.");
    System.out.println("[togglePSJFCurve] : accepts true or false. Optional
argument to toggle the PSJF curve from flat (false) to non-flat (true).");
}

/**
 * @return rand (0,1)
 */
private static double urand() {
    Random rand = new Random();

    return rand.nextDouble();
}

/**
 * @return either arrival time or service time
 */
private static double genexp(double lambda) {
    double u, x;
    x = 0;

    while (x == 0) {
        u = urand();
        x = (-1/lambda)*log(u);
    }
    return x;
}

private static void determineCompletionOrQuantumInterrupt(double
quantumForRR, Clock simulationClock, EventQueue eventQueue, CPU simulationCPU) {
    if (simulationCPU.getMyProcess().getRemainingCpuTime() - quantumForRR <=
0) {
        Event knownCompletion = new Event(EventType.ProcessCompletion,
simulationClock.getSimulationTime() +
simulationCPU.getMyProcess().getRemainingCpuTime());
        eventQueue.insertEvent(knownCompletion);
    }
}

```

```

        } else if (simulationCPU.getMyProcess().getRemainingCpuTime() -
quantumForRR > 0) {
            Event interrupt = new Event(EventType.TimeSliceOccurrence,
simulationClock.getSimulationTime() + quantumForRR);
            eventQueue.insertEvent(interrupt);
        }
    }

    private static void calculateStatistics(SchedulingAlgorithm s, double
totalSimTime, int lambda) throws IOException {
        double cpuUtil = s.cpuUtilization(totalSimTime);
        double avgTurn = s.avgTurnaroundTime(totalSimTime);
        double avgThroughput = s.throughput(totalSimTime);
        double avgProcessInQueue = s.avgProcessesInReadyQueue(lambda);
        double avgWaitingTime = s.avgWaitingTime(totalSimTime);
        // minor correction to rounding
        if (s.getSchedulerType() == SchedulerType.PSJF && cpuUtil > 1) {
            cpuUtil = cpuUtil - 0.0499;
        }

        System.out.println("Average Turnaround Time: " + avgTurn);
        System.out.println("Average Throughput: " + avgThroughput);
        System.out.println("CPU Utilization: " + cpuUtil);
        System.out.println("Average number of processes in Ready Queue: " +
avgProcessInQueue);
        System.out.println("Average Waiting Time: " + avgWaitingTime);

        FileWriter pw = new FileWriter("test.csv", true);
        BufferedReader br = new BufferedReader(new FileReader("test.csv"));
        StringBuilder sb = new StringBuilder();

        if (br.readLine() == null) {
            sb.append("Lambda, Average Turnaround, Throughput, CPU Utilization,
Average # of processes in Ready Queue, Average Waiting Time");
        }

        sb.append('\n');
        sb.append(lambda).append(',');
        sb.append(avgTurn).append(',');
        sb.append(s.throughput(totalSimTime)).append(',');
        sb.append(cpuUtil).append(',');
        sb.append(s.avgProcessesInReadyQueue(lambda)).append(',');
        sb.append(s.avgWaitingTime(totalSimTime));

        pw.write(sb.toString());
        pw.close();
    }
}

```

SchedulingAlgorithm.java

Визначає основний клас для складання алгоритму планування. Містить абстрактне визначення властивостей і поведінки, які поділяють всі планувальники.

```

/**
 * @author Jack Shendrikov
 * Abstract definition of both properties and behavior that all schedulers
share.
 */
public abstract class SchedulingAlgorithm implements PerformanceMetrics {

```

```

private SchedulerType schedulerType;
ProcessReadyQueue myQueue;

static double runningTurnaroundSum = 0;
static double runningBurstTimeSum = 0;
static double runningWaitTimeSum = 0;

// default constructor to be overwritten by specialization classes PSJF, RR
SchedulingAlgorithm() {}

// implement methods from interface as required
@Override
public double avgTurnaroundTime(double totalSimTime) {
    return runningTurnaroundSum / 10000;
}
@Override
public double throughput(double totalSimTime) {
    return 10000 / totalSimTime;
}
@Override
public double cpuUtilization(double totalSimTime) {
    return runningBurstTimeSum / totalSimTime;
}
@Override
public double avgProcessesInReadyQueue(int lambda) {
    return lambda * (runningWaitTimeSum / 10000);
}
@Override
public double avgWaitingTime(double totalSimTime) {
    return runningWaitTimeSum / 10000;
}

SchedulerType getSchedularType() {
    return schedulerType;
}

void setSchedulerType(SchedulerType schedulerType) {
    this.schedulerType = schedulerType;
}

Process getNextProcessForCPU() {
    return myQueue.returnAndRemoveHeadProcess();
}

Process safelyPeekAtNextProcess() { return myQueue.peek(); }

void addProcessToReadyQueue(Process p) {
    myQueue.insertProcess(p);
}
}

```

RR.java

Клас, що визначає Round Robin, успадкований від абстрактного класу Scheduling Algorithm:

```

/****
 * @author Jack Shendrikov
 *
 * Round Robin specialization class that inherits from abstract Scheduling
 * Algorithm
 */

```



```

class RR extends SchedulingAlgorithm {
    RR() {
        this.setSchedulerType(SchedulerType.RR);
        myQueue =
ProcessReadyQueue.createSingleProcessReadyQueueInstance(SchedulerType.RR.getSchedulerType());
    }
}

```

PSJF.java

Клас, що визначає PSJF, успадкований від абстрактного класу Scheduling Algorithm:

```

/**
 * @author Jack Shendrikov
 *
 * Preemptive Shortest Job First specialization class
 */
class PSJF extends SchedulingAlgorithm {
    PSJF() {
        this.setSchedulerType(SchedulerType.PSJF);
        myQueue =
ProcessReadyQueue.createSingleProcessReadyQueueInstance(SchedulerType.PSJF.getSchedulerType());
    }
}

```

PerformanceMetrics.java

Інтерфейс, який надає певний шаблон, значення з якого повинні будуть розраховувати всі планувальники. Вимагає постійного оновлення певних проміжних даних (з використанням значень, отриманих при обробці кожного процесу) протягом усього моделювання.

```

/**
 * @author Jack Shendrikov
 */
public interface PerformanceMetrics {

    double avgTurnaroundTime(double totalSimTime);

    double throughput(double totalSimTime);

    double cpuUtilization(double totalSimTime);

    double avgProcessesInReadyQueue(int lambda);

    double avgWaitingTime(double totalSimTime);
}

```

Process.java

Містить загальні getter'и та setter' для роботи з процесом, містить також значення, які мають розраховуватись для кожного процесу (час початку, час завершення, час повернення, час очікування, час прибуття remainingCpuTime –

використовується для відстеження прогресу процесу на ЦП і того, чи ми можемо вважати його завершеним чи ні), ці дані будуть необхідні як при обробці процесу, так і при обчисленні загальної статистики моделювання.

```
/**
 * @author Jack Shendrikov
 */

class Process {
    private double arrivalTime;    // same as the `Event` ProcessArrival
    eventTime.
    private double burstTime;      // obtained by passing 1/avgServiceTime as
    the lambda in genexp(lambda)
    private double completionTime; //
    private double waitingTime;    //
    private double turnaroundTime; // = completionTime - startTime
    private double startTime;      // = clock when given to CPU
    private double remainingCpuTime; // initialized to burst time and is then
    used to track the process's progress on the CPU and whether we may consider it
    complete or not.
    private boolean isReturning;    //
    private double restartTime;    //

    Process() {
        this.isReturning = false;
    }

    /* Getters and Setters */
    double getArrivalTime() {
        return arrivalTime;
    }
    void setArrivalTime(double arrivalTime) {
        this.arrivalTime = arrivalTime;
    }

    double getBurstTime() {
        return burstTime;
    }
    void setBurstTime(double burstTime) {
        this.burstTime = burstTime;
    }

    double getCompletionTime() {
        return completionTime;
    }
    void setCompletionTime(double completionTime) {
        this.completionTime = completionTime;
    }

    double getWaitingTime() {
        return waitingTime;
    }
    void setWaitingTime(double waitingTime) {
        this.waitingTime = waitingTime;
    }

    double getTurnaroundTime() {
        return turnaroundTime;
    }
    void setTurnaroundTime(double turnaroundTime) {
        this.turnaroundTime = turnaroundTime;
    }

    double getStartTime() {
```

```

        return startTime;
    }
    void setStartTime(double startTime) {
        this.startTime = startTime;
    }

    double getRemainingCpuTime() {
        return remainingCpuTime;
    }
    void setRemainingCpuTime(double remainingCpuTime) {
        this.remainingCpuTime = remainingCpuTime;
    }

    boolean isReturning() {
        return isReturning;
    }
    void setIsReturning(boolean returning) {
        this.isReturning = returning;
    }

    double getRestartTime() {
        return restartTime;
    }
    void setRestartTime(double restartTime) {
        this.restartTime = restartTime;
    }
}

```

ProcessReadyQueue.java

Цей клас поводитьсь як статичний, оголошений *final*, щоб запобігти розширенню класу. Приватний конструктор запобігає створення екземпляра клієнтським кодом, оскільки ми не хочемо створювати екземпляр. Всі функції класу є статичними, оскільки клас не може бути створений, тому не можна викликати методи екземпляра або звертатися до полів екземпляра.

Черга готовності процесу реалізована по-різному в залежності від планувальника.

- 1) Якщо PSJF - використовуємо час, що залишився для сортування черги.
- 2) Якщо RR - новий процес додається у хвіст черги. Якщо процес не завершився в межах виділеного йому кванту, то його робота примусово переривається і він переміщується в хвіст черги. Після закінчення кванта з голови виходить наступний процес.

Також клас містить метод *iterateAndGetRemainingDifferenceForRR*, цей метод необхідний для правильного розрахунку RR використання ЦП. Причина в тому, що після $\lambda = 16,667$ ($1 / 0,06 = 16,667$) черга готовності копіюється з надто великої кількості процесів, що надходять. Round Robin починає вироджуватися і надає кількісні послуги деяким процесам, які ніколи не встигають завершитися. Але необхідно враховувати обсяг одержуваного ними обслуговування ЦП. Тут ми просто додаємо його назад в чисельник, щоб отримати правильний результат.

```

import java.util.LinkedList;
import java.util.PriorityQueue;
import java.util.Queue;

/**
 * @author Jack Shendrikov
 */

final class ProcessReadyQueue {

    // serves as single instance of PRQ to be used throughout simulation
    private static ProcessReadyQueue obj = null;

    private static Queue<Process> schedulerPriorityQueue;

    // private Constructor will prevent the instantiation of this class directly
    private ProcessReadyQueue(int schedulerType) {
        createProcessReadyQueue(schedulerType);
    }

    /**
     * @return the instantiated object of our process ready queue, the only one
     permitted
     */
    static ProcessReadyQueue createSingleProcessReadyQueueInstance(int
schedulerType) {
        // this logic will ensure that no more than one object can be created at
a time
        if (obj == null) {
            obj = new ProcessReadyQueue(schedulerType);
        }
        return obj;
    }

    private void createProcessReadyQueue(int schedulerType) {
        // PSJF
        if (schedulerType == 1) {
            ProcessRemainingTimeComparator PSJFComparator = new
ProcessRemainingTimeComparator();
            schedulerPriorityQueue = new PriorityQueue<>(10, PSJFComparator);
        }

        // RR
        else if (schedulerType == 2) {
            schedulerPriorityQueue = new LinkedList<>();
        }
    }

    void insertProcess(Process p) {
        // adds to queue or if List, to end of list
        schedulerPriorityQueue.add(p);
    }

    Process returnAndRemoveHeadProcess() {
        // retrieve and remove head
        return schedulerPriorityQueue.poll();
    }

    boolean isEmpty() {
        return schedulerPriorityQueue.isEmpty();
    }

    Process peek() {

```

```

        return schedulerPriorityQueue.peek();
    }

    void iterateAndGetRemainingDifferenceForPSJF(double finalTime) {
        for(Process p : schedulerPriorityQueue) {
            Simulator.numProcessesHandled++;
            p.setCompletionTime(finalTime);
            double completionMinusStart = p.getCompletionTime() -
p.getStartTime();
            p.setTurnaroundTime(p.getCompletionTime() - p.getArrivalTime());
            p.setWaitingTime((p.getStartTime() - p.getArrivalTime())
                + (completionMinusStart - p.getBurstTime()));
            if (p.isReturning()) {
                SchedulingAlgorithm.runningBurstTimeSum += p.getBurstTime();
            }
            SchedulingAlgorithm.runningTurnaroundSum += p.getTurnaroundTime();
            SchedulingAlgorithm.runningWaitTimeSum += p.getWaitingTime();
        }
    }

    void iterateAndGetRemainingDifferenceForRR() {
        double workPerformed;
        for(Process p : schedulerPriorityQueue) {
            if (p.isReturning()) {
                workPerformed = p.getBurstTime() - p.getRemainingCpuTime();
                SchedulingAlgorithm.runningBurstTimeSum += workPerformed;
            }
        }
    }
}

```

ProcessRemainingTimeComparator.java

Використовується планувальником PSJF. Порівнює процеси за залишковим CPUTime, тому процес із найменшим часом, що залишився, є першим у черзі.

```

import java.util.Comparator;

/**
 * @author Jack Shendrikov
 *
 * Used by PSJF scheduler.
 */
public class ProcessRemainingTimeComparator implements Comparator<Process> {

    @Override
    public int compare(Process p1, Process p2) {
        return Double.compare(p1.getRemainingCpuTime(),
p2.getRemainingCpuTime());
    }
}

```

CPU.java

CPU отримує процес і:

- встановлює час початку (*startTime*), якщо це перший раз, коли процес обслуговується;

- встановлює прапорець процесу *isReturning* з false на true, щоб наступного разу він був ідентифікований як процес, що повертається.

- якщо процесор працює з процесом, він встановлює свій прапорець *isBusy* в значення true.

```
/**
 * @author Jack Shendrikov
 *
 * CPU receives a process and:
 * - sets the start time if this is the first time the process has been
 serviced;
 * - setting process' boolean flag `isReturning` from false to true, so the
 next time it would be ID'd as a returning process.
 * - if CPU is working on a process, it sets its boolean flag `isBusy` to
 true.
 */

public class CPU {

    private boolean isBusy;
    private Process myProcess;

    CPU() {
        isBusy = false;
    }

    boolean isBusy() {
        return isBusy;
    }

    void setBusy(boolean busy) {
        isBusy = busy;
    }

    Process getMyProcess() {
        return myProcess;
    }

    void setMyProcess(Process myProcess) {
        this.myProcess = myProcess;
    }
}
```

Clock.java

Clock використовується для всієї симуляції, і рахує, власне, час симуляції.

В цьому класі надається простий setter, що викликається в основному циклі управління симулятором. *Clock time* отримується з *EventQueue* (тобто з найранішньої надійшовшої події, яка буде наступною, що буде оброблятися послідовно).

```
/**
 * @author Jack Shendrikov
 */

class Clock {
    private static double simulationTime;

    double getSimulationTime() {
```

```

        return simulationTime;
    }

    void setSimulationTime(double _simulationTime) {
        simulationTime = _simulationTime;
    }
}

```

Event.java

Визначає, що являє собою подію, і засвоює час, в який вона відбувається.

```

/****
 * @author Jack Shendrikov
 */
public class Event {

    private EventType eventType;
    private Double eventTime;

    Event(EventType eventType, double eventTime) {
        this.eventType = eventType;
        this.eventTime = eventTime;
    }

    double getEventTime() {
        return eventTime;
    }

    EventType getEventType() {
        return eventType;
    }

    public void setEventTime(double eventTime) {
        this.eventTime = eventTime;
    }

    @Override
    public String toString() {
        return eventType.toString() + " at time " + eventTime.toString();
    }
}

```

EventTimeComparator.java

Тут події порівнюються за часом прибуття (яке згодом використовується для встановлення поля «час прибуття» (arrivalTime) в *Process.java*), щоб вони могли оброблятися послідовно і підтримувати правильне значення *Clock* для часу моделювання.

```

import java.util.Comparator;

public class EventTimeComparator implements Comparator<Event> {

    @Override
    public int compare(Event e1, Event e2) {
        return Double.compare(e1.getEventTime(), e2.getEventTime());
    }
}

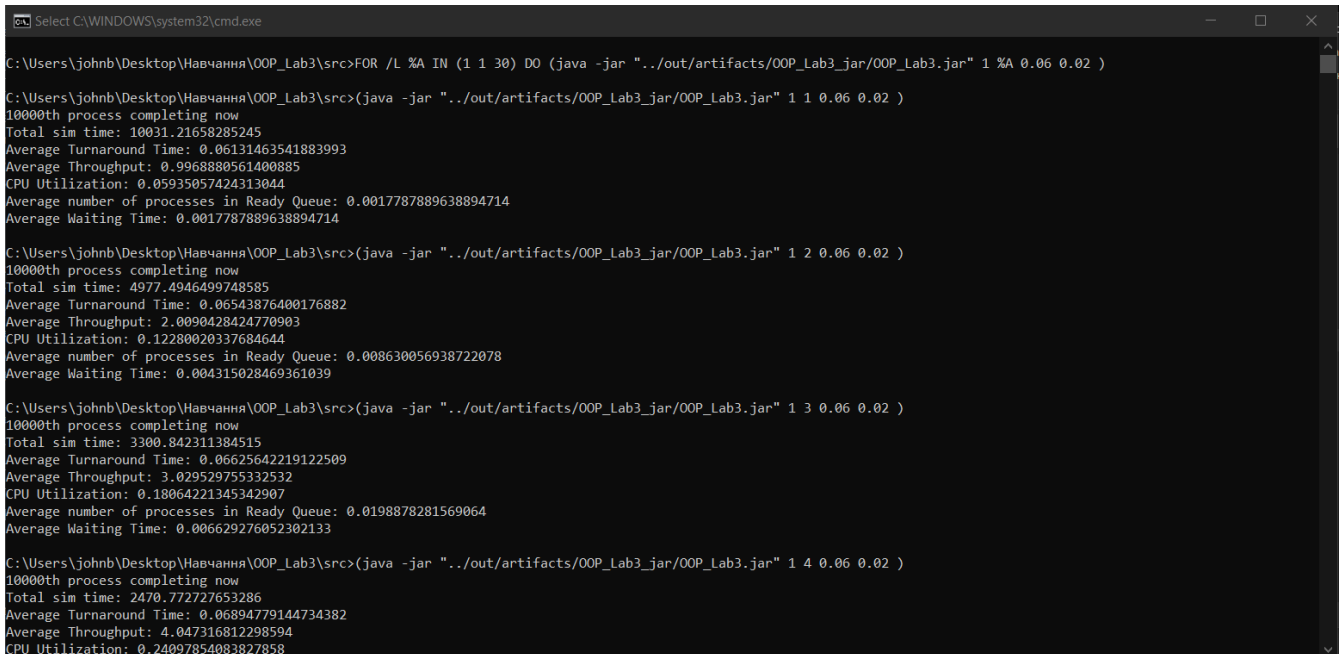
```

Результати виконання роботи

Запуск bat-скрипта (аргументи в секундах, опис аргументів наведено при описі класу *Simulator.java*):

```
FOR /L %%A IN (1,1,30) DO (
    java -jar "../out/artifacts/OOP_Lab3_jar/OOP_Lab3.jar" 1 %%A 0.06 0.02
)
```

Генерація даних:



```
Select C:\WINDOWS\system32\cmd.exe

C:\Users\johnb\Desktop\Навчання\OOP_Lab3\src>FOR /L %A IN (1 1 30) DO (java -jar "../out/artifacts/OOP_Lab3_jar/OOP_Lab3.jar" 1 %A 0.06 0.02 )

C:\Users\johnb\Desktop\Навчання\OOP_Lab3\src>(java -jar "../out/artifacts/OOP_Lab3_jar/OOP_Lab3.jar" 1 1 0.06 0.02 )
10000th process completing now
Total sim time: 10031.21658285245
Average Turnaround Time: 0.06131463541883993
Average Throughput: 0.99688880561400885
CPU Utilization: 0.05935057424313044
Average number of processes in Ready Queue: 0.0017787889638894714
Average Waiting Time: 0.0017787889638894714

C:\Users\johnb\Desktop\Навчання\OOP_Lab3\src>(java -jar "../out/artifacts/OOP_Lab3_jar/OOP_Lab3.jar" 1 2 0.06 0.02 )
10000th process completing now
Total sim time: 4977.4946499748585
Average Turnaround Time: 0.06543876400176882
Average Throughput: 2.0090428424770903
CPU Utilization: 0.12280020337684644
Average number of processes in Ready Queue: 0.008630056938722078
Average Waiting Time: 0.004315028469361039

C:\Users\johnb\Desktop\Навчання\OOP_Lab3\src>(java -jar "../out/artifacts/OOP_Lab3_jar/OOP_Lab3.jar" 1 3 0.06 0.02 )
10000th process completing now
Total sim time: 3300.842311384515
Average Turnaround Time: 0.06625642219122509
Average Throughput: 3.029520755332532
CPU Utilization: 0.18064221345342907
Average number of processes in Ready Queue: 0.0198878281569064
Average Waiting Time: 0.006629276052302133

C:\Users\johnb\Desktop\Навчання\OOP_Lab3\src>(java -jar "../out/artifacts/OOP_Lab3_jar/OOP_Lab3.jar" 1 4 0.06 0.02 )
10000th process completing now
Total sim time: 2470.772727653286
Average Turnaround Time: 0.06894779144734382
Average Throughput: 4.047316812298594
CPU Utilization: 0.24097854083827858
```

Після генерації даних (lambda 1-30) буде сформований .csv файл, на основі якого ми зможемо будувати графіки та проводити порівняльний аналіз.

Найбільш різюча різниця полягає в тому, як обробляється PSJF. Якщо залишити модель як є, деякі криві стануть пологими, що вказує на можливий «ГОЛОД».

Якщо ми включимо процеси, які залишаються в черзі готовності, PSJF стане «найкращим» планувальником, який добре працює під навантаженням.

Якщо залишити модель з параметром `toggleSRTFCurve = false`, середній оборот і середня кількість процесів у черзі готовності будуть практично однаковими в порівнянні з іншими планувальниками.

Якщо ми розглянемо процеси, що залишилися в черзі готовності, то PSJF отримає розподіл, який вказує, що він вигідно відрізняється від інших. Щоправда, здається, що PSJF може страждати від «голоду». Короткі процеси обслуговуються, в той час як інші застряють в черзі готовності. Я використовую суворе визначення завершення, якщо так можна сказати, а саме: процес завершується тільки в тому випадку, якщо він отримує обслуговування ЦП, тривалість якого дорівнює запитаному *burst time*.

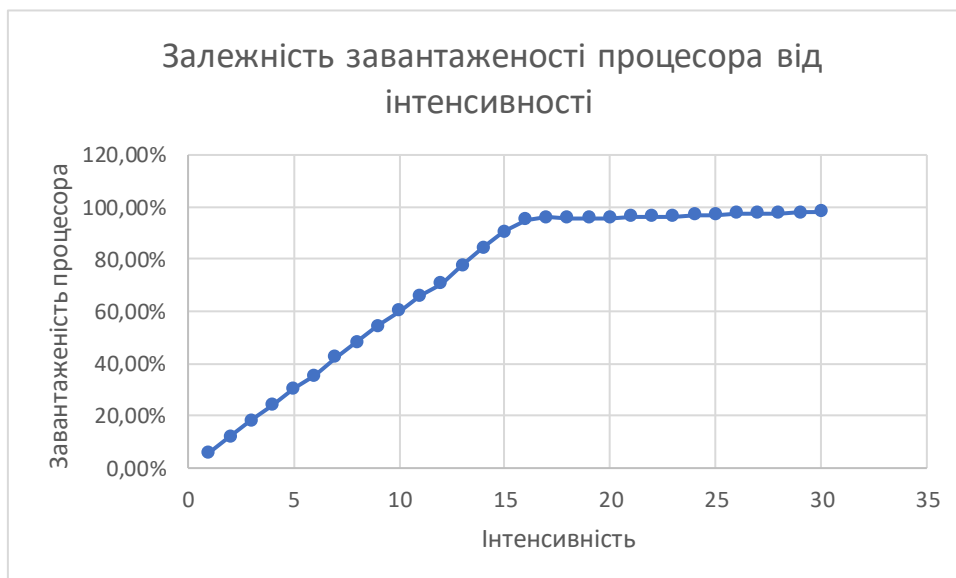
Також при тестуванні було помічено, що Round Robin з квантом 0,01 трохи перевершує RR1 при великому навантаженні. Я припускаю, що це пов'язано з тим, що RR2 може швидко обслуговувати процеси з коротким пакетним часом і відправляти їх по шляху, в той час як RR1 може витратити більше часу на обслуговування більш тривалих процесів.

В цілому, ми можемо підтвердити, що планувальник Round Robin із занадто довгим квантом може виродитися в FCFS.

PSJF— середній час обслуговування 0.06, неплоска крива PSJF.

RR – середній час обслуговування 0.06, квант 0.01 (10мс).

PSJF



Залежність кількості процесів від часу очікування

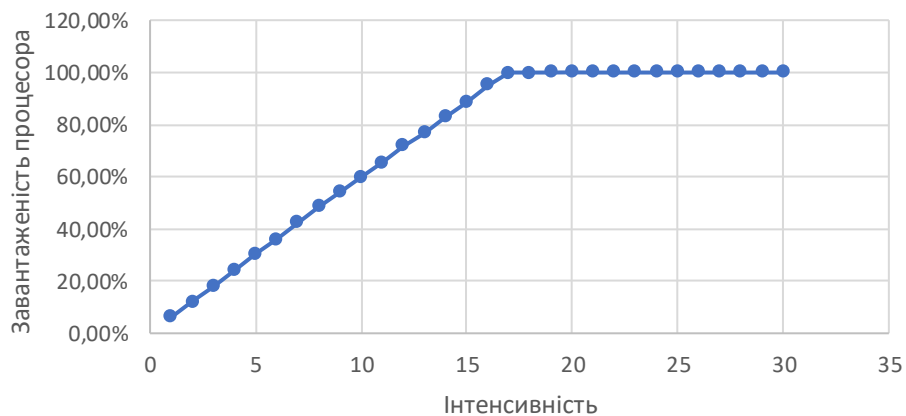


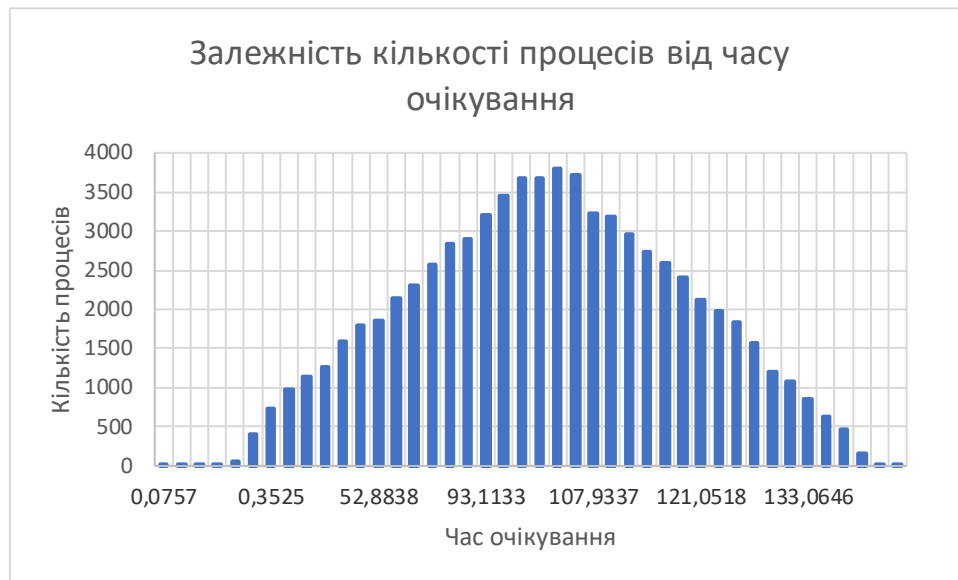
Round Robin

Залежність середнього часу очікування від інтенсивності



Залежність завантаженості процесора від інтенсивності





Я припускаю, що Round Robin з квантом 0,01 є більш ефективним планувальником в порівнянні з іншими заснованими на результатах попереднього тестування. PSJF, в залежності від підходу та інтерпретації, може бути або більш ефективним планувальником за RR, або може страждати від «голоду». Тобто, якщо ми не внесемо ніяких змін в те, як ми обробляємо процеси, скоріш за все, що PSJF страждатиме від «голоду» і не може бути життєздатним варіантом, якщо ми не призначимо процесам, які довго знаходяться в черзі готовності більш високий пріоритет, щоб вони також могли отримати обслуговування в якийсь момент. Але цей момент не був реалізований в межах цієї лабораторної.

Переваги та недоліки досліджуваних дисципліни обслуговування

Переваги Round Robin:

- ✓ При виконанні цього алгоритму планування певний часовий квант розподіляється на різні завдання.
- ✓ За середнім часом відгуку цей алгоритм дає найкращу продуктивність.
- ✓ За допомогою цього алгоритму всі завдання отримують справедливий розподіл ресурсів ЦП.
- ✓ У цьому алгоритмі немає проблем з голодом.
- ✓ Цей алгоритм працює з усіма процесами без будь-якого пріоритету.
- ✓ Цей алгоритм носить циклічний характер.
- ✓ При цьому новостворений процес додається в кінець черги готовності.
- ✓ Крім того, в цьому планувальнику, як правило, використовується розподіл часу, що означає надання кожному процесу часового інтервалу або кванту.

Недоліки **Round Robin**:

- Цей алгоритм витрачає більше часу на перемикання контексту.
- Для невеликого кванта це трудомістке планування.
- Цей алгоритм пропонує більший час очікування і час відгуку.
- У ньому низька пропускна здатність.
- Якщо квант часу занадто малий для планування, то його діаграма Ганта буде занадто велика.

Переваги **PSJF**:

Основною перевагою алгоритму PSJF є те, що він робить обробку завдань швидшою, ніж алгоритм SJF.

Недоліки **PSJF**:

У PSJF перемикання контексту виконується набагато частіше, ніж в SJN, через більшу витрату часу процесора на обробку. Потім витрачений час процесора додає час його обробки, що зменшує перевагу швидкої обробки цього алгоритму.