Национальный технически университет Украины «Киевский политехнический институт»

Факультет информатики и вычислительной техники

Кафедра вычислительной техники

Лабораторная работа №3

По дисциплине «Системное программное обеспечение»

Выполнил: студент 3-го курса,

гр. ИО-01 Соколовский С.

Киев -2013

Вариант = 3

1. Алгоритм FIFO без пріоритетів

**Опис роботи дисципліни обслуговування**

FIFOє найбільш простою стратегією планування процесів і полягає в тому, що ресурс передається тому процесу, котрий раніше всіх інших звернувся до нього. Коли процес потрапляє в чергу готових процесів, process control block приєднується до хвоста черги. Середній час очікування для стратегії FIFO є часто досить великим і залежить від порядку надходження процесів в чергу готових процесів.

Стратегії FIFO притаманний так званий “ефект конвою”. В тому випадку, коли в комп'ютері є один великий процес та декілька малих, то всі процеси збираються на початку черги готових процесів, а згодом в черзі до обладнання. Таким чином, “ефект конвою” призводить до зниженняпропускної здатності як процесору, так і периферійного пристрою.

Листинг программы

import java.util.ArrayList;

import java.util.LinkedList;

import java.util.Queue;

public class Processor {

private int operatedRequestCount;

private int sumWaitTime;

private int currentTime;

private int freeProcessorTime;

private Queue<Request> queue;

public Processor(){

freeProcessorTime=0;

operatedRequestCount=0;

sumWaitTime=0;

queue=new LinkedList<Request>();

}

public void addRequest(Request request){

if(request==null){

throw new IllegalArgumentException();

}

request.setTimeAddingToQueue(currentTime);

queue.add(request);

}

public void step(int count){

currentTime+=count;

Request currentRequest=queue.peek();

if(currentRequest==null){

freeProcessorTime+=count;

return;

}

if(!currentRequest.isOnProcessor()){

currentRequest.startOperateAtTime(currentTime);

}

int dt=currentTime-currentRequest.getTimeRemovedFromQueue();

if(dt>=0){

queue.poll();

operatedRequestCount++;

sumWaitTime+=currentRequest.getTimeRemovedFromQueue()-currentRequest.getTimeAddingToQueue();

}else{

currentRequest.workedTime+=count;

}

if(dt>0){

step(dt);

}

}

public CountTime requestCountWaitTime(){

if(queue.size()==0){

return new CountTime(0, 0);

}

int summaryWaitTime=0;

int currentWaitTime=0;

for(int i=queue.size()-1;i>=0;i--){

currentWaitTime+=((LinkedList<Request>)queue).get(i).getOperateTime();

summaryWaitTime+=currentWaitTime;

}

return new CountTime(queue.size(), summaryWaitTime);

}

public double getAvverageWaitTime(){

return ((double)sumWaitTime)/operatedRequestCount;

}

public double getFreeProcessorPercent(){

return ((double)freeProcessorTime)/currentTime;

}

public int getProcessorTime(){

return currentTime;

}

public String[] requestsToText(){

if(queue.size()==0){

return null;

}

int p=0;

String[] text=new String[queue.size()];

for(int i=queue.size()-1;i>=0;i--){

text[i]=((LinkedList<Request>)queue).get(i).toString();

p++;

}

return text;

}

}

**public** **class** Request {

**private** **int** operateTime;

**private** **int** timeAddingToQueue;

**private** **int** startOperateTime;

**private** **int** timeRemovedFromQueue;

**public** **int** workedTime;

**private** **boolean** isOnProcessor;

**public** Request(**int** operateTime) {

isOnProcessor=**false**;

**this**.operateTime = operateTime;

workedTime=0;

}

**public** **void** startOperateAtTime(**int** processorTime){

isOnProcessor=**true**;

startOperateTime=processorTime;

timeRemovedFromQueue=processorTime+operateTime;

}

**public** **int** getOperateTime() {

**return** operateTime;

}

**public** **void** setOperateTime(**int** operateTime) {

**this**.operateTime = operateTime;

}

**public** **int** getTimeAddingToQueue() {

**return** timeAddingToQueue;

}

**public** **void** setTimeAddingToQueue(**int** timeAddingToQueue) {

**this**.timeAddingToQueue = timeAddingToQueue;

}

**public** **int** getTimeRemovedFromQueue() {

**return** timeRemovedFromQueue;

}

**public** **void** setTimeRemovedFromQueue(**int** timeRemovedFromQueue) {

**this**.timeRemovedFromQueue = timeRemovedFromQueue;

}

**public** **boolean** isOnProcessor() {

**return** isOnProcessor;

}

**public** **void** setOnProcessor(**boolean** isOnProcessor) {

**this**.isOnProcessor = isOnProcessor;

}

**public** String toString(){

**return** "Time adding="+timeAddingToQueue+" operate time="+operateTime+" worked time="+workedTime;

}

}

**public** **class** StreamOperator {

**public** Processor processor;

**private** **int** minOperateTime;

**private** **int** maxOperateTime;

**private** **int** minTimeBetweenRequest;

**private** **int** maxTimeBetweenRequest;

**private** **int** timeToNextRequest;

**public** StreamOperator(**int** minOperateTime,

**int** maxOperateTime, **int** minTimeBetweenRequest,

**int** maxTimeBetweenRequest) {

timeToNextRequest=0;

**this**.processor = **new** Processor();

**this**.minOperateTime = minOperateTime;

**this**.maxOperateTime = maxOperateTime;

**this**.minTimeBetweenRequest = minTimeBetweenRequest;

**this**.maxTimeBetweenRequest = maxTimeBetweenRequest;

}

**private** **int** generateRandom(**int** min,**int** max){

**double** r=Math.*random*();

**return** (**int**)(min+r\*(max-min));

}

**public** **void** modelingSteps(**int** time){

**if**(time<1){

**throw** **new** IllegalArgumentException();

}

Request request=**null**;

**int** operateTime;

**for**(**int** i=0;i<time;i++){

**if**(timeToNextRequest<0){

operateTime=generateRandom(minOperateTime, maxOperateTime);

request=**new** Request(operateTime);

processor.addRequest(request);

timeToNextRequest=generateRandom(minTimeBetweenRequest, maxTimeBetweenRequest);

}**else**{

timeToNextRequest--;

}

processor.step(1);

}

}

**public** **int** getMinOperateTime() {

**return** minOperateTime;

}

**public** **void** setMinOperateTime(**int** minOperateTime) {

**this**.minOperateTime = minOperateTime;

}

**public** **int** getMaxOperateTime() {

**return** maxOperateTime;

}

**public** **void** setMaxOperateTime(**int** maxOperateTime) {

**this**.maxOperateTime = maxOperateTime;

}

**public** **int** getMinTimeBetweenRequest() {

**return** minTimeBetweenRequest;

}

**public** **void** setMinTimeBetweenRequest(**int** minTimeBetweenRequest) {

**this**.minTimeBetweenRequest = minTimeBetweenRequest;

}

**public** **int** getMaxTimeBetweenRequest() {

**return** maxTimeBetweenRequest;

}

**public** **void** setMaxTimeBetweenRequest(**int** maxTimeBetweenRequest) {

**this**.maxTimeBetweenRequest = maxTimeBetweenRequest;

}

}

import java.util.ArrayList;

import javax.swing.JFrame;

import org.jfree.chart.ChartFactory;

import org.jfree.chart.ChartFrame;

import org.jfree.chart.ChartPanel;

import org.jfree.chart.JFreeChart;

import org.jfree.chart.plot.PlotOrientation;

import org.jfree.data.statistics.HistogramDataset;

import org.jfree.data.statistics.HistogramType;

import org.jfree.data.xy.XYDataset;

import org.jfree.data.xy.XYSeries;

import org.jfree.data.xy.XYSeriesCollection;

public class Statistics {

public double[][] waitTimeIntens(int startInt,int endInt,int minOperateTime,

int maxOperateTime,int stepCount){

StreamOperator streamOperator=null;

double[][] data=new double[endInt-startInt][2];

int p=0;

for(int i=startInt;i<endInt;i++){

int st=i-1;

if(st<0){

st=0;

}

streamOperator=new StreamOperator(minOperateTime, maxOperateTime, st, i+1);

streamOperator.modelingSteps(stepCount);

data[p][1]=streamOperator.processor.getAvverageWaitTime();

data[p][0]=i;

System.out.println("x="+data[p][0]+" y="+data[p][1]+" i="+i);

p++;

}

return data;

}

public double[][] freeProcIntens(int startInt,int endInt,int minOperateTime,

int maxOperateTime,int stepCount){

StreamOperator streamOperator=null;

double[][] data=new double[endInt-startInt+1][2];

int p=0;

for(int i=startInt;i<endInt;i++){

streamOperator=new StreamOperator(minOperateTime, maxOperateTime, i-1, i+1);

streamOperator.modelingSteps(stepCount);

data[p][1]=streamOperator.processor.getFreeProcessorPercent();

data[p][0]=i;

p++;

}

return data;

}

public double[][] requestCountWaitTime(int minInt,int maxInt,int minOperateTime,

int maxOperateTime,int stepCount){

StreamOperator streamOperator=new StreamOperator(minOperateTime, maxOperateTime, minInt, maxInt);

CountTime[] res=new CountTime[stepCount];

for(int i=0;i<stepCount;i++){

streamOperator.modelingSteps(1);

res[i]=streamOperator.processor.requestCountWaitTime();

System.out.println(res[i].toString());

}

ArrayList<CountTime> countKey=new ArrayList<CountTime>();

for(int i=0;i<res.length;i++){

CountTime currentCountTime=res[i];

boolean isInList=false;

for(int j=0;j<countKey.size();j++){

if(countKey.get(j).count==currentCountTime.count){

countKey.get(j).time=(countKey.get(j).time+currentCountTime.time)/2;

isInList=true;

break;

}

}

if(!isInList){

countKey.add(res[i]);

}

}

double[][] result=new double[countKey.size()][2];

for(int i=0;i<result.length;i++){

result[i][0]=countKey.get(i).time;

result[i][1]=countKey.get(i).count;

}

return result;

}

public void drawDiagram(double[][] data){

double[] x=new double[data.length];

double[] y=new double[data.length];

for(int i=0;i<data.length;i++){

x[i]=data[i][0];

y[i]=data[i][1];

}

HistogramDataset dataset = new HistogramDataset();

dataset.setType(HistogramType.RELATIVE\_FREQUENCY);

dataset.addSeries("Histogram", x, 40);

String plotTitle = "Histogram";

String xaxis = "number";

String yaxis = "value";

PlotOrientation orientation = PlotOrientation.VERTICAL;

boolean show = false;

boolean toolTips = false;

boolean urls = false;

JFreeChart chart = ChartFactory.createHistogram(plotTitle, xaxis,

yaxis, dataset, orientation, show, toolTips, urls);

int width = 500;

int height = 300;

//try {

//ChartUtilities.saveChartAsPNG(new File("C:\\histogram.PNG"), chart,

// width, height);

//} catch (IOException e) {

//}

ChartPanel chartPanel=new ChartPanel(chart);

JFrame frame=new JFrame();

frame.setContentPane(chartPanel);

frame.setSize(500, 300);

frame.setVisible(true);

}

public void drawGraphic(double[][] data,String name,String xName,String yName){

XYSeries series = new XYSeries(name);

for(int i=0;i<data.length;i++){

series.add(data[i][0],data[i][1]);

}

XYDataset xyDataset = new XYSeriesCollection(series);

JFreeChart chart = ChartFactory.createXYLineChart(

name, xName, yName,

xyDataset, PlotOrientation.VERTICAL, true, true, false);

ChartFrame frame1 = new ChartFrame(name, chart);

frame1.setVisible(true);

frame1.setSize(600, 600);

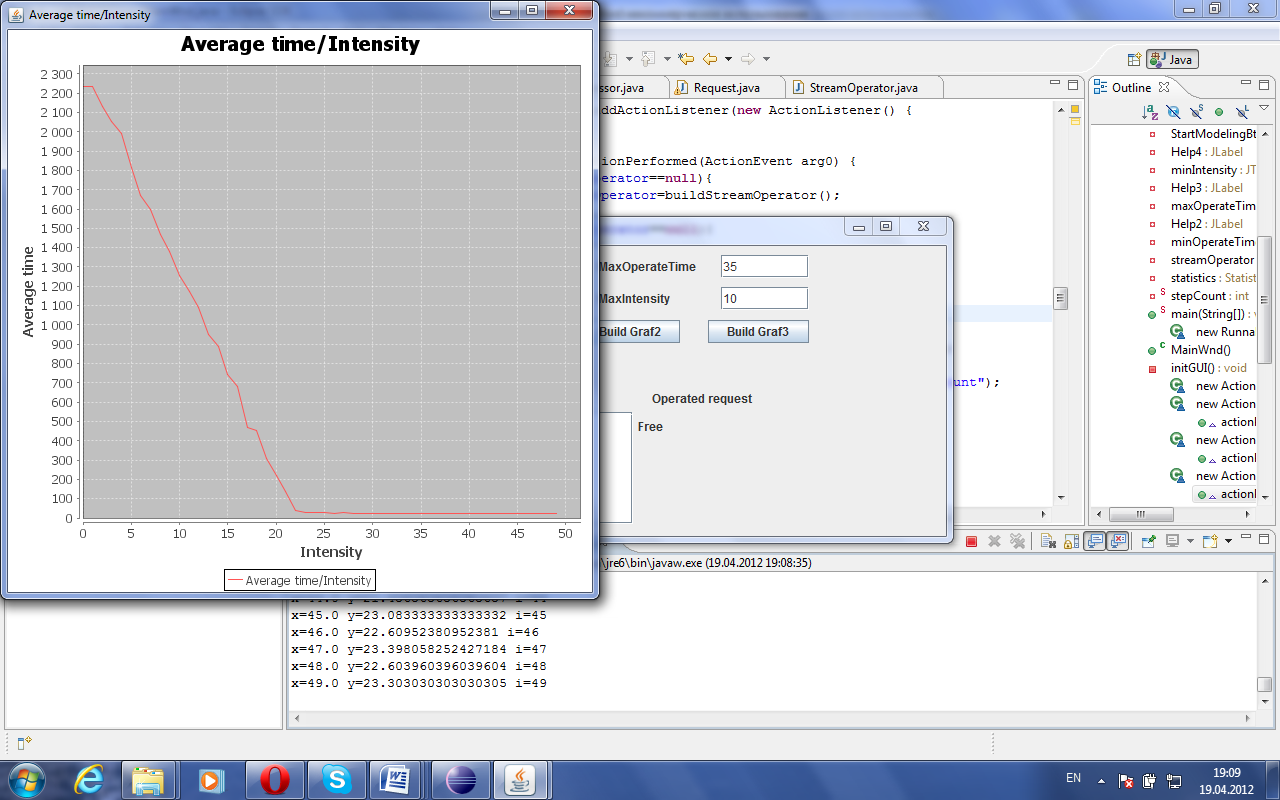
}

}

**Результаты работы**

На всех графиках интенсивность – это время между двумя постпулениями заявок.

**График зависимости времени ожидания, от интенсивности входного потока**



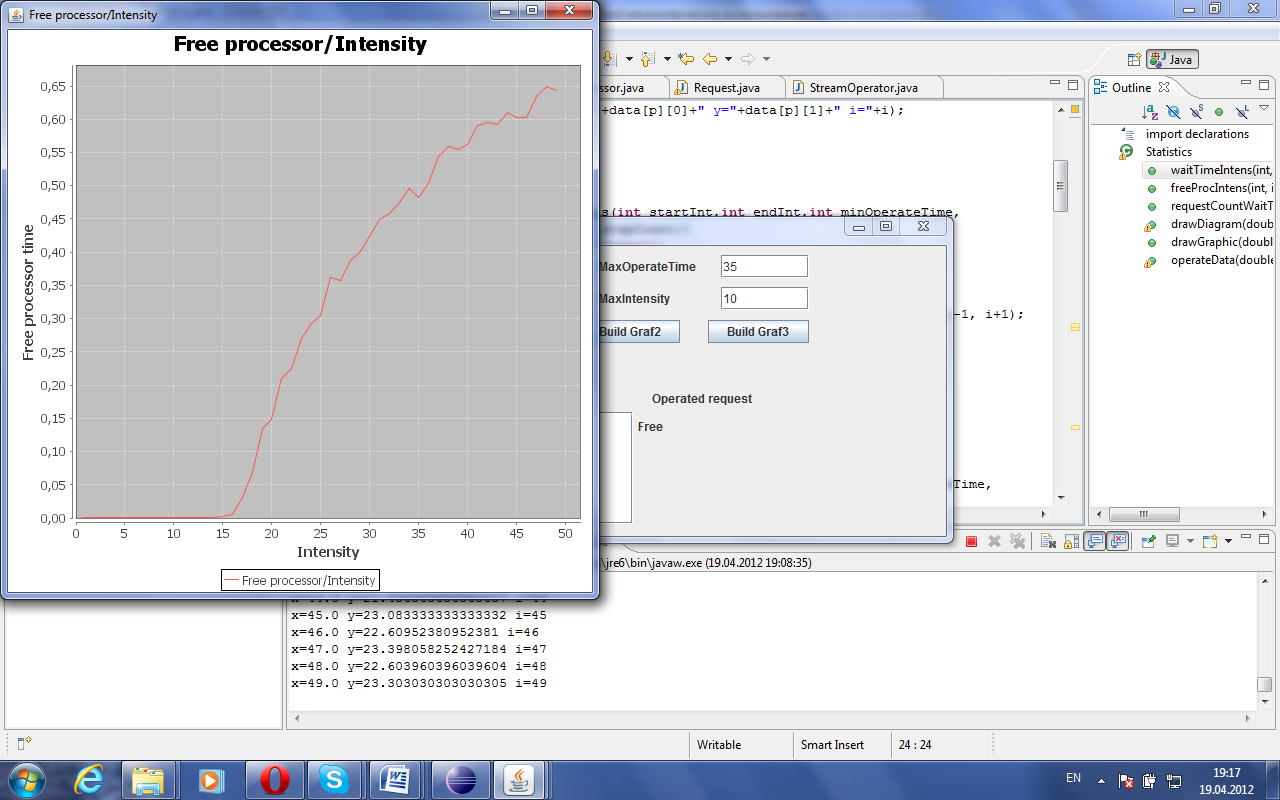
Время обслуживания заявки 10-35

Интенсивность входного потока 0-50

На графике видно, что при небольшом промежутке между входными заявками время обслуживание большое, поскольку заявки не успевают обрабатываться и очередь растет.

При времени между заявками 10+35/2=22.5 заявка успевает обработатся пока не пришла новая заявка, потому заявка практически не находиться в очереди, то есть время обслуживания заявки равно количеству процессорного времени, которое необходимо заявке.

График зависимости простоя от интенсивности входного потока

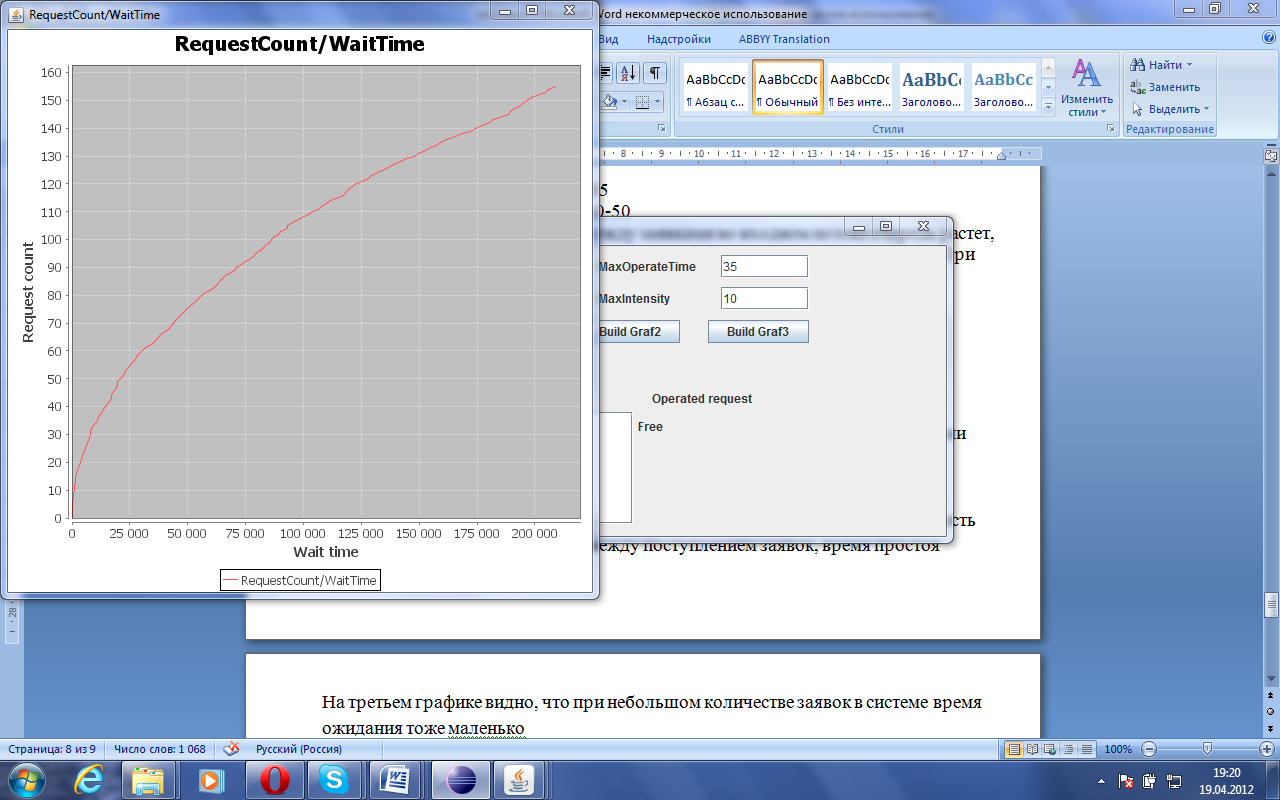


Время обслуживания заявки 10-35

Интенсивность входного потока 0-50

Поскольку при малом времени между заявками во входном потоке очередь растет, то есть постоянно есть заявки для выполнения, то процент простоя равен 0. При увеличении времени между заявками коэффициент простоя растет.

График зависимости количества заявок в системе от времени ожидания



На первом графике видно, что при небольшом количестве заявок в системе суммарное время ожидание тоже не большое, с ростом количества заявок суммарное время тоже растет.

**Достоїнства та недоліки досліджуваної дисципліни обслуговування.**

Преимуществом данной системы обслуживания является ее простота. Недостатком - эффект конвоя.