

Universidad Veracruzana

Facultad de Estadística e Informática

Sistemas Operativos

Proyecto algoritmo de planificación de proceso SFJ

Nicole Moreno Hernández

11/12/2023

Contenido

[Introducción 3](#_Toc153241361)

[¿Cómo funciona el algoritmo de planificación de proceso Shortest Job First? 3](#_Toc153241362)

[Descripción del algoritmo: 4](#_Toc153241363)

[¿Cómo se realizó? 7](#_Toc153241364)

[Librerías 7](#_Toc153241365)

[Clase proyecto 8](#_Toc153241366)

[Clase Proceso 9](#_Toc153241367)

[Clase ProcesoMenor 10](#_Toc153241368)

[Clase HiloGeneradorProceso 11](#_Toc153241369)

[Clases HiloAdministradorProcesos 12](#_Toc153241370)

[Interfaz Gráfica 12](#_Toc153241371)

[Código 12](#_Toc153241372)

[Explicación 14](#_Toc153241373)

[Conclusión: 17](#_Toc153241374)

# Introducción

En un entorno de multiprogramación, coexisten procesos, trabajos e hilos que se ejecutan simultáneamente. Sin embargo, el sistema operativo se enfrenta a limitaciones, ya que algunos recursos no pueden reasignarse automáticamente y otros no pueden compartirse entre trabajos. Además, existe un número finito de recursos, como la memoria.

Con el fin de abordar estas limitaciones y alcanzar objetivos como maximizar el rendimiento, la eficiencia y minimizar el tiempo de respuesta y de rotación, se han desarrollado algoritmos de planificación de procesos.

Algunos de estos algoritmos incluyen "First Come, First Served" (FCFS), "Shortest Job Next" (SJN), algoritmo de prioridad, "Shortest Remaining Time" (SRT), "Round Robin", colas de múltiples niveles y "Earliest Deadline First" (EDF). Estos algoritmos juegan un papel crucial en la gestión efectiva de los recursos y la programación eficiente de procesos en sistemas multiprogramados.

Algunos de los algoritmos son más eficientes que otros, por ejemplo, el Roud Robin es el algoritmo más utilizado, sin embargo, queda a consideración del programador, cuál de ellos utilizar.

En este documento se presentará una pequeña simulación con interfaz de como funciona el algoritmo de planificación de proceso Shortest Job First o también conocido como Shortest Job Next, de igual manera, se mostrará el motivo de la implementación y como se fue construyendo los códigos.

# ¿Cómo funciona el algoritmo de planificación de proceso Shortest Job First?

Es un algoritmo que se basa en la longitud de el tiempo de ciclo que necesita en la CPU, no es preemptivo y no suele funcionar en todos los sistemas interactivos, pues además de que el tiempo no es algo de que se conozca muy fácilmente, puede haber proceso que nunca se ejecutaran, ya que siempre pueden llegar procesos más pequeños.

Ejemplo:

Tenemos 4 proceso:

* Proceso A con ciclos en la CPU de 7
* Proceso B con ciclos en la CPU de 14
* Proceso C con ciclos en la CPU de 9
* Proceso D con ciclos en la CPU de 3

La línea en la que se ejecutarían estos procesos es: D, A, C y B.

B

C

A

D

# Descripción del algoritmo:

El lenguaje utilizado fue java, por medio del software NetBeans.

**Código:**

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

/\*\*

\*

\* @author nmh14

\*/

public class Proyecto {

public static void main(String args[]) {

InterfazGrafica interfaz = new InterfazGrafica();

ProcesoMenor procesoX = new ProcesoMenor();

HiloGeneradorProceso hiloGP = new HiloGeneradorProceso(procesoX, interfaz);

HiloAdministradorProcesos hiloAP = new HiloAdministradorProcesos(procesoX, interfaz);

interfaz.setVisible(true);

hiloGP.start();

hiloAP.start();

}

}

class Proceso {

private int id;

private int memoria;

private int tiempo;

public Proceso(int id, int tiempo, int memoria) {

this.id = id;

this.memoria = memoria;

this.tiempo = tiempo;

}

public void setId(int id) {

this.id = id;

}

public void setMemoria(int memoria) {

this.memoria = memoria;

}

public void setTiempo(int tiempo) {

this.tiempo = tiempo;

}

public int getId() {

return id;

}

public int getMemoria() {

return memoria;

}

public int getTiempo() {

return tiempo;

}

}

class ProcesoMenor {

private ArrayList<Proceso> listaProcesos;

public ProcesoMenor() {

this.listaProcesos = new ArrayList<>();

}

public synchronized void agregarProceso(Proceso p) {

listaProcesos.add(p);

Collections.sort(listaProcesos, Comparator.comparingInt(Proceso::getTiempo));

}

public synchronized Proceso getProceso() {

if (!listaProcesos.isEmpty()) {

Proceso p = listaProcesos.get(0);

listaProcesos.remove(0);

return p;

}

return null;

}

public synchronized ArrayList<Proceso> getLista() {

return new ArrayList<>(listaProcesos);

}

}

class HiloGeneradorProceso extends Thread {

private ProcesoMenor procesoMenor;

private InterfazGrafica interfaz;

public HiloGeneradorProceso(ProcesoMenor p, InterfazGrafica interfaz) {

this.procesoMenor = p;

this.interfaz = interfaz;

}

@Override

public void run() {

int id = 0;

while (true) {

int tiempo = (int) (Math.random() \* 1000 + 1);

int memoria = (int) (Math.random() \* 525 + 2);

Proceso proceso = new Proceso(id, tiempo, memoria);

procesoMenor.agregarProceso(proceso);

try {

Thread.sleep(1000);

} catch (Exception e) {

}

System.out.println("Proceso generado-ID: " + id + ", Tiempo: " + tiempo + ", Memoria: " + memoria);

interfaz.actualizarListaProcesos(procesoMenor.getLista());

interfaz.actualizarProcesoActual(proceso);

id++;

}

}

}

class HiloAdministradorProcesos extends Thread {

private ProcesoMenor pMenor;

private InterfazGrafica interfaz;

HiloAdministradorProcesos(ProcesoMenor p, InterfazGrafica i) {

this.pMenor = p;

this.interfaz = i;

}

public void procesar() {

while (true) {

Proceso proceso = pMenor.getProceso();

if (proceso != null) {

interfaz.mostrarProcesoEnEjecucion("Proceso " + proceso.getId() + " con tiempo de " + proceso.getTiempo() + " en ejecución");

try {

Thread.sleep(proceso.getTiempo() \* 10);

} catch (Exception e) {

}

System.out.println("Proceso " + proceso.getId() + " completado.");

}

}

}

@Override

public void run() {

while (true) {

procesar();

}

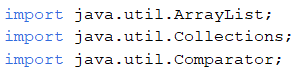
}

}

# ¿Cómo se realizó?

## Librerías

La clase donde se desarrolló todo el proyecto se llamó Proyecto, y se utilizaron 3 librerías:



* import java.util.ArrayList;

Está librería se agregó para poder utilizar los métodos que tiene por defecto Arraylist, como add(), get() y remove().

* import java.util.Collections;

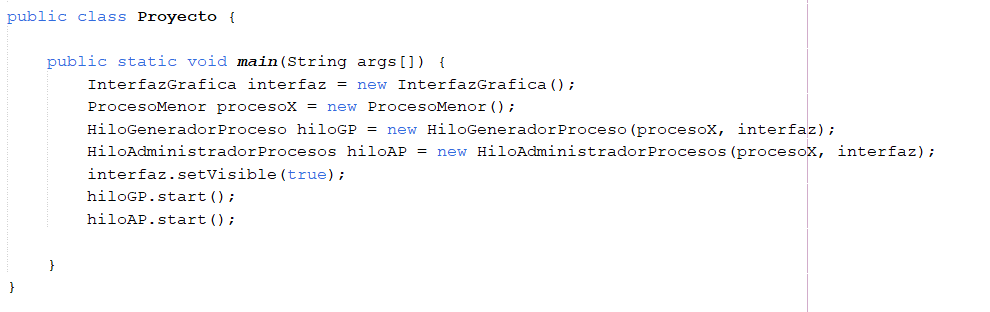
Está librería se uso para ordenar la lista de manera ascendente, se puede utilizar para listas, conjuntos (set) y mapeado (map), se urilizí el método sort().

* import java.util.Comparator;

Esta librería se utilizó para definir que el ordenamiento sea entre números enteros en función a la variable tiempo.

## Clase proyecto

La clase proyecto contiene el main:



Donde se genera una interfaz grafica para cuando comience a correr el programa, esta se haga visible y también se pueda ver en la interfaz como trabaja el simulador.

Tiene la variable procesoX, que representa un número de proceso en general.

Tiene la inicialización del hilo que genera proceso, así como el hilo que administra los proceso, con los atributos procesoX y de la interfaz.

Tiene el comando para que la interfaz sea visible.

Y por ultimo tienen los start() que es para correr ambos hilos.

## Clase Proceso



Posteriormente tiene la clase Proceso las cuales tiene las variables enteras, id, tiempo y memoria, tiene sus respectivos getters y setters.

## Clase ProcesoMenor



La clase ProcesoMenor contiene la inicialización de la ArrayList listaProcesos, creada para guardar todos los procesos generados.

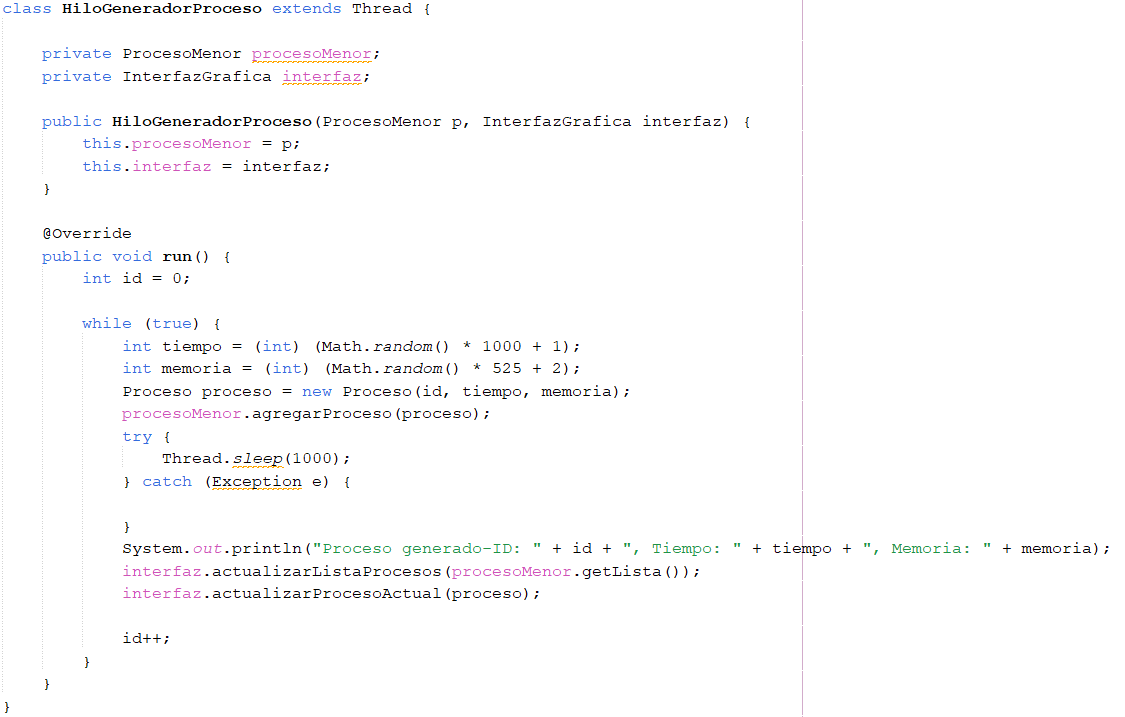
El método agragarProceso() se apoya del comando add(), que se puede utilizar gracias a la librería de ArrayList, este ingresa los proceso a la listaProcesos, además con el método de ordenación Collection, permite ordenar de manera ascendente los elementos de lista procesos, comparando con el método Comporator, que compara los enteros (hace referencia con .comparingInt) obtiene el tiempo con getTiempo de la clase Proceso

El método getProceso() verifica si la lista no está vacia, sino es así, guarda en una variable de tipo Proceso, el primer proceso de la lista, (se obtiene con get(0)) ya que se sabe que están ordenados, así que el primero siempre será el más chico, posteriormente lo elimina de la lista, ya que se supone que una ves que entra al administrador de proceso, ya no necesita volver ni guardar esa información para después, esto se hace como el comando remove(0). Si es el caso de que la lista está vacía, entonces solo retorna nulo.

Sobre el método getLista(), devuelve la lista listaProceso.

Los 3 anteriores métodos son sincronizados para que no exista ningún problema como de carrera entre los hilos, y cada hilo pueda ejecutar el hilo a la vez, a pesar de que se pudo haber reemplazado con un semáforo, por la manera en la que se utilizan los métodos (todo el método prácticamente debería estar dentro del semáforo) consideré más conveniente utilizar synchronized.

## Clase HiloGeneradorProceso



Está clase contiene los parámetros procesoMenor de la clase procesoMenor e interfaz de la clase InterfazGrafica

En el método run() se encuentra la generación de hilos, encabezada por la declaración de la variable id, ya que está por cada iteración va a aumentar uno, posteriormente se encuentra un ciclo while(true) es decir, siempre se va a ejecutar. Se declaran las variables tiempo y memoria, que por cada iteración son diferentes ya que se declaran para que sean random, tiempo, puede tener un tiempo entre 1 y 1000 mientras que memoria un número entre 2 entre 525, representando que la mayor memoria que puede necesitar un proceso para ejecutarse es toda la memoria disponible, un dato que comentar es que, no se tomó en cuenta el caso en el que la memoria pudiera ser mayor a la memoria disponible para que se ejecute el proceso, ya que el algoritmo no hace hincapié en la memoria.

Una ves que se generar las variables, se crea un objeto de la clase proceso con los parámetros, posteriormente se agrega a la lista con agregarProceso(proceso).

Este hilo está hecho para que se genere un hilo cada segundo.

Los procesos agregados se añaden a la tabla de la interfaz y se señalan con actualizarListaProceso y actualizarProcesoActual, por último, el id incrementa uno.

## Clases HiloAdministradorProcesos



Al igual que la anterior clase, tiene los mismos atributos inicializados, sin embargo, en esta ocasión procesoMenor se llama pMenor.

Tiene un solo método que es el de procesar, contiene un ciclo while(true) es decir siempre se va a ejecutar, de el procesoMenor obtiene el proceso de la posición cero, o sea el que tiene el tiempo más pequeño, lo escribe en el área de texto de la interfaz, que indica cual es el proceso que entró a administración de procesos.

De ahí se duerme el hilo durante el tiempo que dura el proceso por 10, se iba a simular por 1000, pero eran demasiados segundos, por lo que se prefirió bajar a 10

Por ultimo se encuentra el método run() que nuevamente tiene el ciclo while(true) y lo que se está ejecutado es el proceso

# Interfaz Gráfica

# Código

import javax.swing.\*;

import java.awt.\*;

import java.util.ArrayList;

import javax.swing.table.DefaultTableModel;

/\*\*

\*

\* @author nmh14

\*/

public class InterfazGrafica extends JFrame {

private JTable tablaProcesos;

private DefaultTableModel tablaMuestraP;

private DefaultListModel<String> modeloLista;

private JList<String> listaProcesos;

private JLabel etiquetaProcesoActual;

private JTextArea txtPEjecucion;

public InterfazGrafica() {

setTitle("Simulador de Proceso de SJF");

setSize(600, 500);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

etiquetaProcesoActual = new JLabel("Procesos en espera");

tablaMuestraP = new DefaultTableModel();

tablaProcesos = new JTable(tablaMuestraP);

txtPEjecucion = new JTextArea();

txtPEjecucion.setEditable(false);

tablaMuestraP.addColumn("ID");

tablaMuestraP.addColumn("Tiempo");

tablaMuestraP.addColumn("Memoria");

JScrollPane scrollPane = new JScrollPane(tablaProcesos);

JPanel contenedorTabla = new JPanel(new BorderLayout());

contenedorTabla.add(etiquetaProcesoActual, BorderLayout.NORTH);

contenedorTabla.add(scrollPane, BorderLayout.CENTER);

contenedorTabla.add(txtPEjecucion, BorderLayout.SOUTH);

add(contenedorTabla);

}

public void actualizarListaProcesos(ArrayList<Proceso> procesos) {

tablaMuestraP.setRowCount(0);

for (Proceso proceso : procesos) {

tablaMuestraP.addRow(new Object[]{proceso.getId(), proceso.getTiempo(), proceso.getMemoria()});

}

}

public void actualizarProcesoActual(Proceso proceso) {

int filaProcesoActual = getProcesoNuevo(proceso.getId());

if (filaProcesoActual >= 0) {

tablaProcesos.setRowSelectionInterval(filaProcesoActual, filaProcesoActual);

}

}

private int getProcesoNuevo(int idProceso) {

for (int i = 0; i < tablaMuestraP.getRowCount(); i++) {

Object valorCelda = tablaMuestraP.getValueAt(i, 0);

if (valorCelda != null && Integer.parseInt(valorCelda.toString()) == idProceso) {

return i;

}

}

return -1;

}

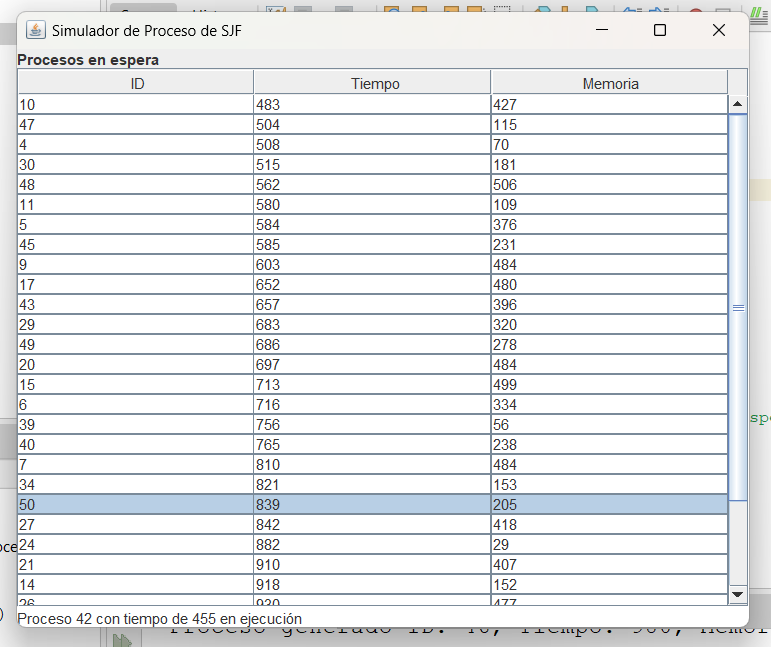
public void mostrarProcesoEnEjecucion(String infoProceso) {

txtPEjecucion.setText(infoProceso);

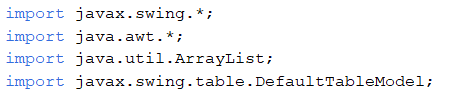
}

}

# Explicación



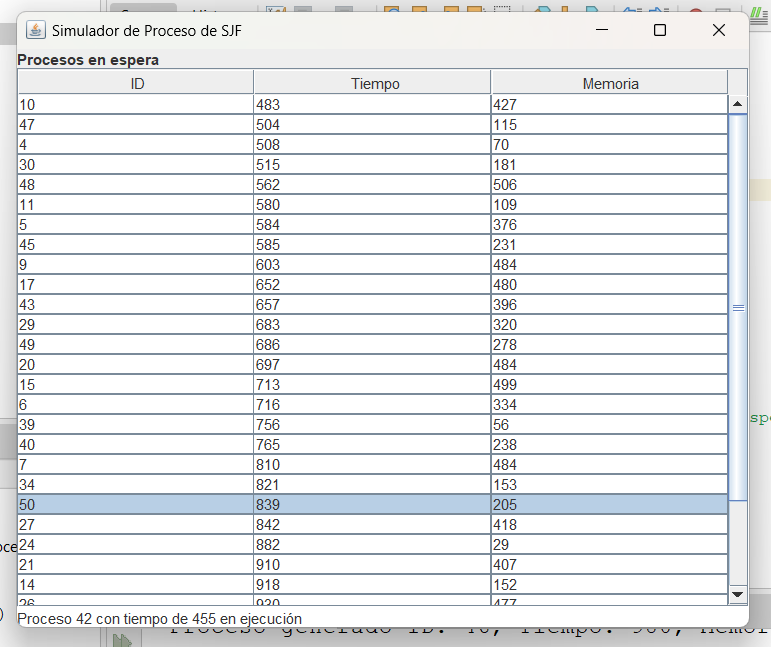
El código se irá explicando conforme las partes de la interfaz.

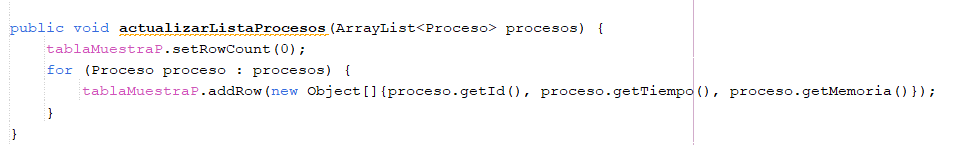


Las librerías utilizadas son las básicas para la creación de interfaces gráficas en java, la única diferencia es la librería de ArrayList para agregar los elementos a la tabla.



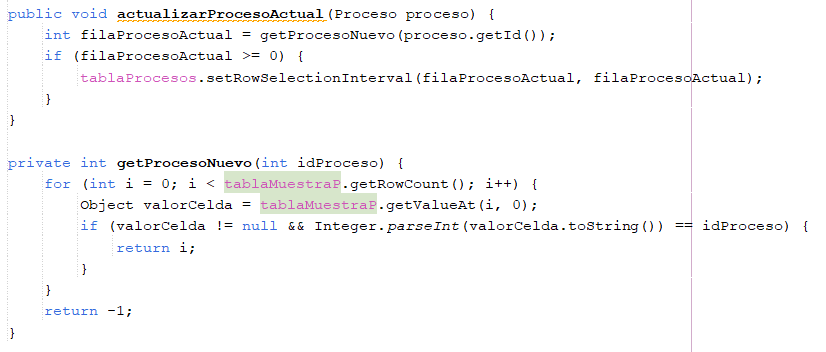
En esta parte se inicializa todo lo básico sin proceso, la tabla junto son sus columnas, el titulo de la pestaña, el tamaño de la ventana, se agrega el área de texto y se agrega una barra para bajar y subir si la tabla se hace más grande, es decir lo señalado a continuación.



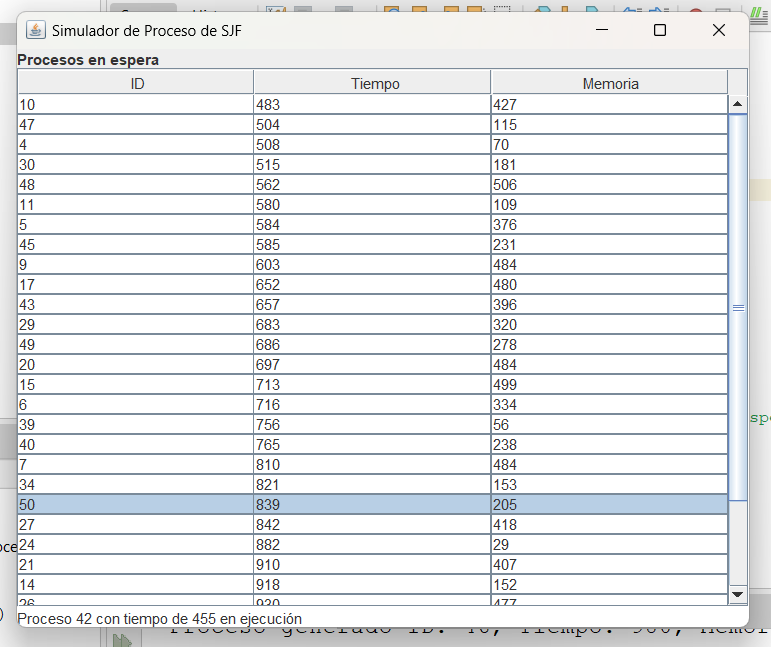


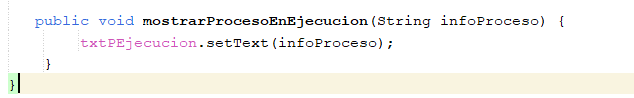
Este método agrega toda lista completa, para empezar, borra todas las filas, ya que, sino las borra se van a agregar toda la lista, generando que en la tabla se muestren 2 proceso con id 2 y así aumentado por cada iteración.

Con el for se agrega toda la lista guardada.

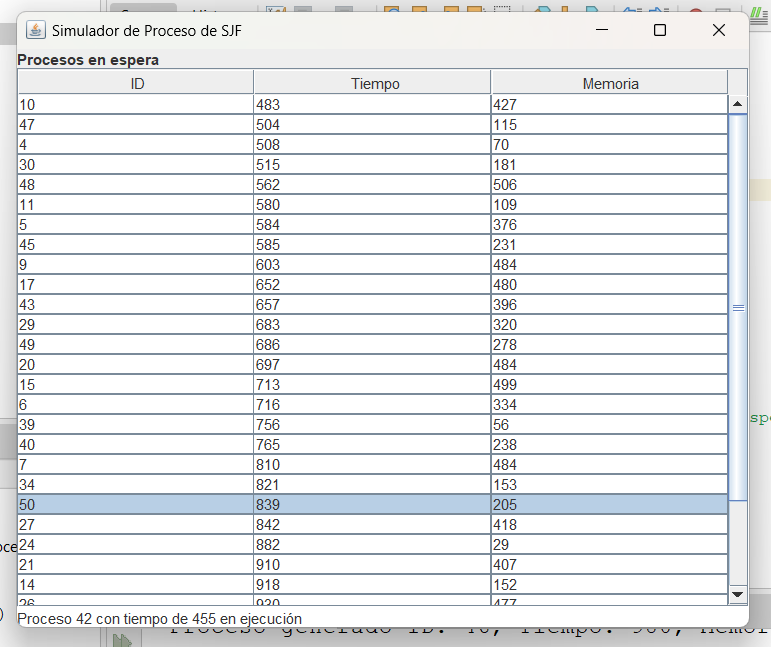


Ambos métodos se ocupan para seleccionar el nuevo método ingresado y se selecciona, en el caso de getProcesoNueve convierte el id y en string y busca si la fila con ese id tiene el mismo id que el nuevo proceso, si es así envía el número de columna a actulizarProcesoActual y si este no es -1 de que no se encontró, entonces va a marca la fila entera del proceso nuevo agregado





Este proceso solo se encarga de actualizar el proceso que ha entrado en el administrador de procesos.



# Conclusión:

Los hilos son una manera muy eficaz de generar procesos y poder trabajar con ellos, sin embargo, hay que ser muy cuidadosos, ya que deben tener un control, en cuanto al algoritmo, si que es cierto que su codificación no fue la más difícil, sin embargo como ya se comento anteriormente, no es la más eficiente, porque en los momentos que hacia pruebas, pude observar que había procesos que nunca se iban a ejecutar, porque habían proceso nuevos que se generaban cada segundo y habían muchos otros procesos que iban antes de él.

Lo que más se complicó de la codificación considero que fue la interfaz gráfica y la manera en la que iba a simular el proceso, pues había pensado en ir restando tiempo y se lo agregaba hasta que fuera menor que 0, ahí salía de ese ciclo y otro ciclo nuevo podía entrar, pero se me dificultó implementarlo ya que tenía que jugar con los setters y dos ciclos, fue más sencillo dormir al hilo y removerlo de la lista.

En cuento la interfaz, fue difícil ya que esperaba trabajar con un jframe form, pero fue demasiado difícil, así que decidí hacerlo a mano.