Universidad Veracruzana

**Facultad De Estadística e Informática.**

**Ingeniería De Software.**

**Proyecto Individual – Segundo Parcial.**

**Sistemas Operativos.**

**Alumno:**

**Velázquez Chicuellar Carim.**

**Docente:**

**López Herrera Juan Luis**

# PROYECTO INDIVIDUAL – SEGUNDO PARCIAL

A continuación, en la presente documentación voy a redactar y explicar de manera breve la funcionalidad de mi programa que aborda mi proyecto individual, así como la lógica implementada para su comprensión, esperando cumplir con todo o al menos con la mayoría de los requerimientos solicitados.

Antes que nada, empecemos entendiendo el algoritmo que me corresponde y que se implementó en mi programa, siendo éste el algoritmo de **Planificación por Prioridad (Priority Scheduling)**, el cuál simple y sencillamente se encarga de asignar la CPU según la prioridad más alta que se asigne a cada proceso en la cola, por ende, el proceso con la mayor prioridad más alta se ejecutará primero. Como es lógico, los procesos en este caso se clasifican según su prioridad y se ejecutan en orden de mayor a menor prioridad.

1. Empecemos con lo primero que son las librerías implementadas, que fueron las siguientes:
   a.
   ```
   import javax.swing.*;
   import java.awt.*;
   import java.util.*;
   import java.util.List;
   ```
   b. En el caso de *"import java.util.List;"* fue un caso especial debido a que hice uso de una lista ligada (*LinkedList<>()*) y si no la ponía me marcaba error (lo explicaré más a fondo más adelante).

2. Ahora empecemos con las clases implementadas, que en mi caso la primera fue **Proceso:**
   a.
   ```
   class Proceso{
       private static int contador_id = 0;
       private int id;
       private int tiempo;
       private int memoria;
       private int prioridad;

       public Proceso(){
           this.id = ++contador_id;
           Random random = new Random();
           this.tiempo = random.nextInt(bound:10) + 1;
           this.memoria = random.nextInt(bound:100) + 1;
           this.prioridad = random.nextInt(bound:5) + 1;
       }

       public int getId(){
           return id;
       }

       public int getTiempo(){
           return tiempo;
       }

       public int getMemoria(){
           return memoria;
       }

       public int getPrioridad(){
           return prioridad;
       }
   }
   ```

b. Explicación de las líneas:
    i. El **contador estático** se utiliza para asignar identificadores únicos a cada instancia de la clase **Proceso**. Es estático para que mantenga su valor entre todas las instancias de la clase.
    ii. Los demás atributos (**id, tiempo, memoria, prioridad**) representan las características de un proceso. Cada proceso tiene un identificador único (id), un tiempo de ejecución que será aleatorio (tiempo), una cantidad de memoria necesaria que tambien será aleatoria (memoria), y una prioridad de igual manera aleatoria (prioridad).
    iii. En el constructor de la clase nos encargamos de inicializar un nuevo proceso, posteriormente se asigna un identificador único y se generan valores aleatorios para el tiempo, memoria y la prioridad.
    iv. Finalmente, los métodos getters que como ya sabemos son los que nos van a ayudar con la obtención de la información específica sobre los procesos.

3. Ahora pasemos con la siguiente clase que es la clase que simula una **CPU:**

a.
```
class CPU{
    private Proceso proceso_actual;

    public synchronized void asignarProceso(Proceso proceso){
        this.proceso_actual = proceso;
        System.out.println("Proceso " + proceso.getId() + " asignado a la CPU.");
    }

    public synchronized void liberarProceso(){
        System.out.println("Proceso " + proceso_actual.getId() + " liberado.");
        this.proceso_actual = null;
    }
}
```

b. Explicación de las líneas:
    i. El único atributo que hay es el del proceso actual, es cual evidentemente nos representa el proceso que está actualmente asignado a la CPU. Es privado para encapsular el estado de la CPU.
    ii. El método para asignar procesos nos va a permitir asignar un nuevo proceso a la CPU, siendo este un método sincronizado (*synchronized*) para asegurarnos de que un solo un hilo pueda ejecutarlo a la vez, y asi evitar problemas de concurrencia. Finalmente se asigna el proceso (que se pasó por parámetro en la función) a la variables "*proceso_actual*" y muestra un mensaje en la consola.
    iii. Por último, el método "*liberarProceso()*" como su nombre nos dice, nos va a permitir liberar el proceso que está asignado actualmente a la CPU. Al ser igualmente un proceso sincronizado, tambien nos garantiza que un solo hilo pueda ejecutarlo a la vez, y ya al final nos muestra el mensaje en

la consola indicando que el proceso ha sido liberado y se establece el proceso actual como nulo.

4. La siguiente clase que es la clase **Memoria** la explicaré por partes:

a.

```java
class Memoria{
    public Queue<Proceso> obtenerColaProcesos(){
        return cola_de_procesos;
    }

    private int capacidad_maxima;
    private Queue<Proceso> cola_de_procesos;

    public Memoria(int capacidad_maxima){
        this.capacidad_maxima = capacidad_maxima;
        this.cola_de_procesos = new LinkedList<>();
    }

    public synchronized boolean agregarProceso(Proceso proceso){
        try{
            if(cola_de_procesos.size() < capacidad_maxima){
                cola_de_procesos.add(proceso);
                System.out.println("Proceso " + proceso.getId() + " llegó a la memoria.");
                return true;
            }
            else{
                throw new RuntimeException("No hay espacio para el proceso " + proceso.getId() + ".");
            }
        }
        catch(RuntimeException e){
            System.out.println("Error al agregar proceso: " + e.getMessage());
            return false;
        }
    }
```

b. Explicación de estas primeras líneas de codigo (parte 1):
   i. Primero nos encontramos con el método "*obtenerProcesos()*" que como su nombre lo dice, nos va a permitir obtener la cola de procesos de la memoria. Al ser publica la cola, las otras clases podrán tener acceso a ella, pero no deberán modificarla directamente para poder mantener el encapsulamiento. Al final nos retorna la cola de procesos.
   ii. El atributo "*capacidad_maxima*" que es privado nos representa la capacidad máxima que la memoria podrá tener, es decir, cuantos procesos podrá almacenar.
   iii. Ahora, el atributo "*cola_de_procesos*" es el atributo (privado) que representa la cola de los procesos en la memoria del sistema, y se utiliza la estructura de cola (**Queue**) para poder gestionar los procesos que lleguen.
   iv. El constructor de la clase **Memoria** recibe como parámetro la capacidad máxima de memoria y la asigna a su atributo correspondiente. Además, inicializa la cola de procesos con una nueva instancia de *LinkedList*, que como dije al principio, para poder hacer eso debí agregar la librería "import java.util.List;".

5. Parte 2 de la clase **Memoria**, que corresponde al algoritmo de **Planificación por Prioridad (Priority Scheduling):**
   a. 
   ```
   public synchronized Proceso obtenerProceso(){
       Proceso procesoConMayorPrioridad = null;

       for(Proceso proceso : cola_de_procesos){
           if(procesoConMayorPrioridad == null || proceso.getPrioridad() > procesoConMayorPrioridad.getPrioridad()){
               procesoConMayorPrioridad = proceso;
           }
       }

       if(procesoConMayorPrioridad != null){
           cola_de_procesos.remove(procesoConMayorPrioridad);
       }

       return procesoConMayorPrioridad;
   }
}
   ```
   b. Explicación de las líneas:
      i. Primero, el método "*obtenerProceso()*" es el método que implementa el algoritmo de planificación por prioridad. Nos va a devolver el proceso con la mayor prioridad en la cola de procesos declarada anteriormente. Este método de igual manera es sincronizado (synchronized), ya que aca tambien buscamos evitar los problemas de concurrencia cuando muchos hilos intentan acceder a la cola al mismo tiempo.
      ii. Despues, la variable "*procesoConMayorPrioridad*" es la variable que almacenará TEMPORALMENTE el proceso que tenga la mayor prioridad que se haya encontrado durante la iteración.
      iii. En el bucle "for-each" se hará la iteración sobre la cola de procesos, lo cual nos ayudará a recorrer todos los procesos en la cola.
      iv. Ahora, en la comparación de las prioridades, vamos a comparar la prioridad del proceso actual con la prioridad del proceso almacenado en la variable antes declarada "*procesoConMayorPrioridad*". Posteriormente, si el proceso actual tienen una mayor prioridad, se actualiza la variable "*procesoConMayorPrioridad*".
      v. Despues de que hayamos encontrado el proceso con mayor prioridad, se va a eliminar de la cola de procesos.
      vi. Finalmente se retornará el proceso con prioridad mayor (puede que sea null si la cola de procesos llegara a estar vacía).

6. Continuamos con la siguiente clase que es **InterfazGrafica** la cual va a simular el comportamiento grafico de los procesos.

a.

```
class InterfazGrafica extends JFrame{
    private JTextArea logArea;
    private JPanel cpuPanel;
    private JPanel memoriaPanel;

    public InterfazGrafica(){
        setTitle(title:"Simulador de Asignación de CPU y Memoria");
        setSize(width:600, height:400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        logArea = new JTextArea();
        logArea.setEditable(b:false);

        cpuPanel = new JPanel();
        cpuPanel.setBackground(Color.GRAY);

        memoriaPanel = new JPanel();
        memoriaPanel.setLayout(new GridLayout(rows:5, cols:1));
        memoriaPanel.setBackground(Color.LIGHT_GRAY);

        JScrollPane scrollPane = new JScrollPane(logArea);

        add(cpuPanel, BorderLayout.NORTH);
        add(memoriaPanel, BorderLayout.CENTER);
        add(scrollPane, BorderLayout.SOUTH);

        setLocationRelativeTo(c:null);
        setVisible(b:true);
    }
```

b.

```
public void agregarLog(String mensaje){
    logArea.append(mensaje + "\n");
}

public void actualizarCPU(Proceso proceso){
    if(proceso != null){
        cpuPanel.setBackground(Color.RED);
    }
    else{
        cpuPanel.setBackground(Color.GRAY);
    }
}
```

c. Explicación de las líneas (parte 1):

i. Primero se hace la declaración de las variables de instancia “logArea”, “cpuPanel” y “memoriaPanel”, el cual son componentes de la interfaz gráfica.

ii. Despues, en el constructor de la clase básicamente se realiza la configuración de la apariencia y disposición de la interfaz gráfica al probar el programa, donde ajustamos los títulos, tamaños y el cierre de la ventana.

iii. Posteriormente se hace la configuración de los componentes, los cuales son: **logArea** = área de texto para mostrar mensajes de registro, **cpuPanel** = panel que representará el estado de la CPU, inicialmente de color gris, **memoriaPanel** = panel con un "*GridLayout*" de 5 filas y 1 columna que va a representar la memoria y el **scrollPanel** = que será el desplazamiento que envuelve el área de texto.
iv. Continuando con los componentes de la ventana, ahora se configuran los paneles, colocándolos el **cpuPanel** al norte, el **memoriaPanel** al centro y el **scrollPanel** al sur.
v. Ahora, continuando con la configuración de la ventana, ajustamos en el centro la ventana en la pantalla con "setLocationRelativeTo(null)" y la hacemos visible con "setVisible(true)".
vi. Posteriormente con el método "*agregarLog(String mensaje)*" añadimos un mensaje al área de registro.
vii. Finalmente con el método "*actualizarCPU(Proceso proceso)*" actualizamos el colol de panel de la CPU según sea el caso (ROJO = hay un proceso asignado o GRIS = no hay ningún proceso asignado).

7. Continuando con parte 2 de la clase **InterfazGrafica:**

a.
```
    public void actualizarMemoria(Queue<Proceso> cola_de_procesos){
        memoriaPanel.removeAll();

        if(cola_de_procesos != null){
            List<Proceso> lista_de_procesos = new ArrayList<>(cola_de_procesos);

            for(Proceso proceso : lista_de_procesos){
                JPanel procesoPanel = new JPanel();
                procesoPanel.setBackground(Color.blue);
                procesoPanel.setBorder(BorderFactory.createLineBorder(Color.black));
                procesoPanel.setPreferredSize(new Dimension(width:50, height:50));
                memoriaPanel.add(procesoPanel);
            }
        }
        memoriaPanel.revalidate();
        memoriaPanel.repaint();
    }
}
```

b. Explicación de las líneas (parte 2):
   i. Primero, el método "*actualizarMemoria(Queue<Proceso> cola_de_procesos)*" es el que se va a encargar de actualizar la representación visual de la memoria en la interfaz grafica.
   ii. Despues, la linea "memoriaPanel.removeAll()" va a limpiar todos los componentes existentes en **memoriaPanel**, asegurándose de que esté vacío antes de agregar nuevos elementos.
   iii. Ahora, en la sentencia "if (cola_de_procesos != null)" nos aseguramos de que la cola de procesos no sea nula antes de procesarlar.

iv. Posteriormente, hacemos la creación de la lista de procesos con "*List<Proceso> lista_de_procesos = new ArrayList<>(cola_de_procesos)"* el cual convierte la cola de procesos a una lista para hacer más facil su manipulación.
v. En el siguiente bucle: "for (Proceso proceso : lista_de_procesos)" iteramos sobre cada proceso en la lista.
vi. Posteriormente creamos un nuevo panel para representar visualmente el proceso con: "JPanel procesoPanel = new JPanel()".
vii. Establecemos el fondo del panel en color azul con: "procesoPanel.setBackground(Color.blue)".
viii. Añadimos un borde negro alrededor del panel con: "procesoPanel.setBorder(BorderFactory.createLineBorder(Color.black))".
ix. Despues establecemos las dimensiones preferidas del panel con 50, 50.
x. Y tambien agregamos el panel al panel de memoria.
xi. Cabe aclarar que las anteriores líneas fueron simplemente diseño de la ventana y no hay mucha importancia en ellas.
xii. Ahora, para actualizar visualmente el panel de la memoria usamos: "memoriaPanel.revalidate()" que vuelve a validar el panel de memoria para reflejar los cambios y "memoriaPanel.repaint()" para repintar el panel de memoria para que los cambios sean visibles.

8. Consiguientemente, pasamos ahora con la clase **SistemaOperativo** que será la clase que implementará la lógica del sistema operativo simulado.
   a.

```
class SistemaOperativo implements Runnable{
    private CPU cpu;
    private Memoria memoria;
    private InterfazGrafica gui;
    private int ejecuciones_maximas;
    private int ejecuciones_actuales;
    private boolean cpuOcupada;

    public SistemaOperativo(CPU cpu, Memoria memoria, InterfazGrafica gui, int ejecuciones_maximas){
        this.cpu = cpu;
        this.memoria = memoria;
        this.gui = gui;
        this.ejecuciones_maximas = ejecuciones_maximas;
        this.ejecuciones_actuales = 0;
    }

    public boolean cpuEstaOcupada(){
        return cpuOcupada;
    }

    public void controladorEjecuciones(){
        while(ejecuciones_actuales < ejecuciones_maximas){
            try{
                Thread.sleep(millis:100);
            }
            catch (InterruptedException e){
                e.printStackTrace();
            }
            ejecuciones_actuales++;
        }
    }
```

b. Explicación de las líneas (parte 1):
    i. Primero empezamos declarando las variables privadas: **cpu** = de tipo CPU que representa la unidad central de procesamiento, **memoria** = de tipo Memoria que representa la memoria del sistema, **gui** = de tipo InterfazGrafica que representa la interfaz grafica del sistema, **ejecuiones_maximas** = de tipo entera que será para almacenar el número máximo de ejecuciones que realizará el sistema operativo, **ejecuciones_actuales** = de tipo entera que será para llevar el conteo de las ejecuciones actuales del sistema y finalmente la variable **cpuOcupada** = que será para indicarnos si la CPU está actualmente ocupada o no.
    ii. Despues, en el constructor de la clase inicializamos las variables del sistema operativo con los valores que proporcionamos al crear las instancias de las clases.
    iii. Ahora, en el método "*cpuEstaOcupada()*" simplemente nos devuelve **true** si la CPU en efecto está ocupada, y **false** en caso contrario.
    iv. Finalmente, en método "*controlarEjecuciones()*" lo que hace es que controla las ejecuciones del sistema operativo mediante un bucle while, para posteriormente utilizar "*Thread.sleep(n)*" para pausar el hilo durante n milisegundos en cada iteración.
    v. Incrementa la variable **ejecuciones_actuales** en cada iteración hasta que se alcanza el numero de ejecuciones máximas.

9. Continuando con la parte 2 de la clase **SistemaOperativo:**

a.
```
    @Override
    public void run(){
        Thread control_de_hilo = new Thread(this::controladorEjecuciones);
        control_de_hilo.start();

        while(true){
            Proceso proceso = new Proceso();
            if(memoria.agregarProceso(proceso)){

                cpuOcupada = true;
                cpu.asignarProceso(proceso);

                try{
                    Thread.sleep(proceso.getTiempo() * 100);
                }
                catch(InterruptedException e){
                    e.printStackTrace();
                }
            }

            cpu.liberarProceso();
            cpuOcupada = false;

            SwingUtilities.invokeLater(() -> {
                gui.actualizarCPU(proceso);
                gui.actualizarMemoria(memoria.obtenerColaProcesos());
            });

            ejecuciones_actuales++;
        }
    }
}
```

b. Explicación de las líneas (parte 2):
    i. Primero, hacemos la implementación del método **run** de la interfaz **Runnable**, que es la que controla el flujo principal del simulador cuando se inicia un nuevo hilo.
    ii. Despues, en la linea “Thread control_de_hilo = new Thread(this::controladorEjecuciones);” creamos un nuevo hilo que va a ejecutar el método “*controladorEjecuciones*”.
    iii. Posteriormente, iniciamos el hilo creado para controlar la cantidad total de ejecuciones con la linea: “control_de_hilo.start();”
    iv. Ahora, en el buche while (infinito), nos va a representar el funcionamiento continuo del sistema operativo.
    v. Despues de eso, se crea un nuevo objeto **Proceso** en cada iteración, representando un nuevo proceso en el sistema.
    vi. En la sentencia “if(memoria.agregarProceso(proceso))” verificamos si es posible agregar el proceso a la memoria utilizando el método “*agregarProceso*” de la clase Memoria.
    vii. En las siguientes líneas “cpuOcupada = true; cpu.asignarProceso(proceso);” marcamos la CPU como ocupada y se asigna el proceso a la CPU.
    viii. En los bloques **try-catch**, el hilo se va a dormir durante un tiempo determinado representado por “proceso.getTiempo() * 100” milisegundos, sumulando asi, el tiempo de ejecución del proceso.
    ix. Despues liberamos la CPU y la marcamos como no ocupada despues de la ejecución del proceso, todo esto con las líneas “cpu.liberarProceso(); cpuOcupada = false;”.
    x. Finalmente, utilizamos “*SwingUtilities.invokeLater*” para actualizar la interfaz grafica de manera segura, ya que se esta ejecutando en un hilo diferente al de la de la interfaz gráfica. Al final incrementamos el contador de las ejecuciones actuales.

10. Finalmente, pasamos con la clase que contiene el método Main que en mi caso es **Proyecto:**
    a.
```
public class Proyecto{
    Run | Debug
    public static void main(String[] args){
        CPU cpu = new CPU();
        Memoria memoria = new Memoria(capacidad_maxima:10);
        InterfazGrafica gui = new InterfazGrafica();
        SistemaOperativo sistemaOperativo = new SistemaOperativo(cpu, memoria, gui, ejecuciones_maximas:10);
        Thread HiloSO = new Thread(sistemaOperativo);

        HiloSO.start();
    }
}
```
    b. Explicación de las líneas:
        i. En esta clase vamos a emplear el método **main** para finalmente poder ejecutar el programa.

ii. Primero creamos la instancia de la clase CPU llamada **cpu**, el cual va a simular el comportamiento del CPU.
iii. Creamos la instancia de la clase Memoria llamada **memoria**, con una capacidad máxima que el usuario podrá elegir.
iv. Creamos una instancia de la clase InterfazGrafica llamada **gui**, que nos va a representar la interfaz grafica del simulador.
v. Tambien creamos la instancia de la clase SistemaOperativo llamada **sistemaOperativo**, el cual va a controlar la lógica del SO, a la cual se le proporcionará la CPU, memoria, interfaz grafica y el numero entero máximo de ejecuciones.
vi. Finalmente creamos el hilo llamado **HiloSO** que ejecutará la logia del sistema operativo representada por la instancia sistemaOperativo para posteriormente iniciarlo con “start()”.

Y asi es como concluye la explicación y/o documentación de mi proyecto individual del segundo parcial de la materia Sistemas Operativos, esperando haber sido claro y habiendo cumplido con todos los requerimientos que se solicitaron, cabe recalcar que en caso de que me haya olvidado de algo o temas similares, tambien estaré grabando el video con la explicación del codigo.

Por su atención, gracias.