Proyecto 1: Simulador de Planificación de Procesos

Franco Barra Roger Balan

26 de octubre de 2025

Índice

1 Introducción

El objetivo de este proyecto es el desarrollo de un simulador en Java para sistemas monoprocesador que permita comprender y aplicar los conceptos fundamentales de la planificación de procesos. El simulador implementa la gestión completa del ciclo de vida de los procesos, el manejo de excepciones de Entrada/Salida (E/S) y seis políticas de planificación distintas.

A través de la simulación y la recolección de métricas, este informe analiza la configuración óptima del sistema, evaluando el rendimiento (throughput, utilización de CPU, tiempos de respuesta) de cada algoritmo bajo diferentes cargas de trabajo, cumpliendo con los requisitos especificados en el enunciado del proyecto.

2 Funcionalidad de Clases y Métodos Principales

El simulador está diseñado siguiendo un modelo modular donde cada clase tiene una responsabilidad definida, orquestadas por la clase `SistemaOperativo`.

2.1 SistemaOperativo.java

Es la clase central que actúa como el núcleo del simulador. Implementa `Runnable` y contiene el bucle principal de la simulación.

- **Gestión del Reloj:** Administra el `globalClock` y la duración del ciclo (`cycleDuration`), la cual puede ser modificada en tiempo real.
- **Orquestación:** Posee instancias de `CPU`, `Scheduler`, `GestorEstados` y `GestorDeMetricas`, coordinando sus interacciones.
- **Colas Principales:** Administra las colas de `readyQueue`, `blockedQueue`, `terminatedQueue` y `suspendedReadyQueue`.
- **Método `run()`:** Es el hilo principal de la simulación. En cada ciclo, adquiere un semáforo (mutex) para sincronizarse con el `IOManager`, ejecuta el método `_runCycle()` para procesar la lógica del ciclo, y libera el mutex.
- **Método `_runCycle()`:** Contiene la lógica principal del SO: admite nuevos procesos, procesa las interrupciones de E/S, verifica el estado de la CPU y, si está ociosa, llama a `dispatch()`.
- **Método `dispatch()`:** Llama al `scheduler.selectNextProcess()` para obtener el siguiente PCB. Si la política es apropiativa, maneja el desalojo del proceso actual. Finalmente, asigna el nuevo proceso a la CPU.

2.2 PCB.java (Process Control Block)

Representa la estructura de datos fundamental de un proceso. Contiene toda la información necesaria para su gestión:

- **Identificadores:** `id` (único y autoincremental), `name`.
- **Estado:** `status` (enum `ProcessStatus` que incluye NEW, READY, RUNNING, BLOCKED, TERMINATED, SUSPENDED_READY, SUSPENDED_BLOCKED).
- **Registros y Ejecución:** `programCounter`, `longitudPrograma`, `ciclosRestantes`.

- **Tipo de Proceso:** `type` (enum `ProcessType`: CPU_BOUND o IO_BOUND).
- **Parámetros de E/S:** Si es IO_BOUND, almacena `ciclosParaExcepcion` y `ciclosParaSatisfacerIO`.
- **Métricas de Proceso:** Almacena `arrivalTime`, `tiempoFinalizacion`, `tiempoRespuesta`, `tiempoEsperaAcumulado`, etc., que son utilizados por el `GestorDeMetricas`.
- **Método `executeInstruction()`:** Simula la ejecución de una instrucción, decrementando `ciclosRestantes` e incrementando `programCounter`.

2.3 Scheduler.java

Implementa la lógica de selección de procesos. Es el componente clave para intercambiar las políticas de planificación.

- **Enum `SchedulingAlgorithm`:** Define las 6 políticas requeridas: FCFS, SJF_NON_PREEMPTIVE, SJF_PREEMPTIVE, ROUND_ROBIN, PRIORITY_NON_PREEMPTIVE, y PRIORITY_PREEMPTIVE.
- **Método `selectNextProcess()`:** Retorna el siguiente proceso a ejecutar. Para políticas no apropiativas o RR/FCFS, simplemente extrae (`sacar()`) el primer elemento de la `readyQueue`.
- **Método `reinsertProcess(PCB pcb)`:** Es el método más importante. Cuando un proceso entra a la cola de listos (desde nuevo, bloqueado o suspendido), este método lo inserta en la posición correcta según la política activa.
- **Métodos de Inserción Ordenada:** Utiliza *helpers* como `_insertOrderedBySJF()` y `_insertOrderedByPriority()` para mantener la `readyQueue` ordenada, garantizando que `selectNextProcess()` siempre obtenga el PCB correcto. Para FCFS y RR, simplemente añade al final.

2.4 CPU.java

Simula el hardware del procesador como un recurso pasivo.

- **Método `dispatch(PCB pcb)`:** Asigna un proceso a la CPU, cambiando su estado a RUNNING.
- **Método `executeCycle()`:** Es llamado por el `SistemaOperativo` en cada ciclo. Ejecuta `pcb.executeInstruction()` y verifica tres condiciones de interrupción:
  1. Finalización del proceso (`ciclosRestantes <= 0`).
  2. Solicitud de E/S (si es IO_BOUND y `programCounter % ciclosParaExcepcion == 0`).
  3. Fin del Quantum (si `quantumCounter >= quantumSize`).
- **Método `endExecution()`:** Desaloja el proceso de la CPU y retorna el PCB al `SistemaOperativo` para que este decida su siguiente estado.

2.5 GestorEstados.java

Centraliza la lógica de transición de estados, asegurando que los procesos se muevan correctamente entre las colas.

- **Método** `moveBlockedToReady(PCB pcb)`**:** Método clave que, al completarse una E/S, llama a `scheduler.reinsertProcess(pcb)` para colocar el proceso en la `readyQueue` según la política de planificación activa.
- **Gestión de Suspensión:** Implementa `suspendReadyProcess()`, `resumeReadyProcess()`, etc., para mover procesos entre las colas de listos/bloqueados y `suspendedReadyQueue`/`suspendedBlo`

2.6 IOManager.java

Simula un dispositivo de E/S que opera de forma concurrente con la CPU. Implementa `Runnable` para ejecutarse en su propio hilo.

- **Sincronización:** Utiliza un `Semaphore` (actuando como Mutex) para coordinar el acceso a las colas compartidas con el `SistemaOperativo`.
- **Método** `run()`**:** Es el hilo del dispositivo. Adquiere el mutex, llama a `processBlockedQueue()`, libera el mutex y duerme por un tiempo (`ioCycleDuration`), simulando la concurrencia.
- **Método** `processBlockedQueue()`**:** Itera sobre la `blockedQueue` (sin extraer los elementos), decrementa el contador `ciclosIOEspera` de cada PCB. Si un contador llega a 0, el PCB se añade a una cola temporal `readyToMove`.
- **Finalización de E/S:** Al final de la iteración, todos los PCBs en `readyToMove` son extraídos de la `blockedQueue` y pasados al `gestorEstados.moveBlockedToReady()`.

2.7 Estructuras de Datos (Cola.java y Node.java)

Se implementó una estructura de datos de Cola personalizada basada en una lista enlazada simple (`Node`), cumpliendo con el requisito de no utilizar las librerías de colecciones de Java. Provee las operaciones atómicas `agregar(pcb)`, `sacar()` y `verFrente()`.

2.8 Gestión de la Interfaz (GUI.java)

Implementa la interfaz gráfica en Swing.

- **Inicialización:** Es responsable de crear todas las instancias de los componentes del simulador (`CPU`, `Scheduler`, `SO`, `IOManager`, etc.) e iniciar sus respectivos hilos.
- **Actualización Asíncrona:** Utiliza un `javax.swing.Timer` que dispara el método `actualizarUI()` periódicamente. Esto permite refrescar la interfaz de forma segura (en el hilo de Swing) sin bloquear el hilo de la simulación.
- **Interacción:** Los *listeners* de los botones (ej. `btnAnadirProceso`) y ComboBox (`comboAlgoritmo`) capturan la entrada del usuario y llaman a los métodos correspondientes del `SistemaOperativo` (ej. `so.admitirProceso()`, `so.setSchedulingAlgorithm()`).

2.9 Persistencia (ConfiguracionManager.java)

Maneja la carga de la configuración inicial desde un archivo.

- **Método** `cargarConfiguracion()`**:** Lee un archivo `.csv` que especifica la duración del ciclo y la lista de procesos iniciales (nombre, instrucciones, tipo, y parámetros de E/S).

2.10 Métricas (GestorDeMetricas.java)

Clase dedicada a recolectar y calcular las métricas de rendimiento.

- **Recolección:** Acumula los PCBs en la `terminatedQueue` y registra los ciclos de CPU inactiva.
- **Cálculos:** Provee métodos para calcular `Throughput`, `Utilización CPU`, `Tiempo de Retorno Promedio`, `Tiempo de Espera Promedio`, `Tiempo de Respuesta Promedio` y `Equidad`.

3 Conclusiones de las Configuraciones

Esta sección presenta el análisis de rendimiento de las seis políticas de planificación implementadas. Las conclusiones se basan en la data recolectada por el `GestorDeMetricas` bajo diferentes escenarios de simulación.

3.1 Metodología de Pruebas

Para evaluar las políticas, se diseñaron dos escenarios de carga principales, cada uno ejecutado durante 50.000 ciclos de reloj para obtener métricas estables.

- **Escenario 1: Carga CPU-Bound.**
  Se cargaron 10 procesos: 8 procesos **CPU-Bound** (6000 ciclos c/u) y 2 procesos **I/O-Bound** (1000 ciclos c/u, E/S cada 100 ciclos). Este escenario simula un servidor de cómputo científico o renderizado.
- **Escenario 2: Carga I/O-Bound.**
  Se cargaron 10 procesos: 2 procesos **CPU-Bound** (6000 ciclos c/u) y 8 procesos **I/O-Bound** (1000 ciclos c/u, E/S cada 100 ciclos, satisfacción de 200 ciclos). Este escenario simula un servidor web o de base de datos.

Los resultados se comparan utilizando las siguientes métricas clave: Throughput (procesos/10k ciclos), Utilización de CPU (%) y Tiempo de Respuesta Promedio (ciclos).

3.2 Análisis de Métricas por Algoritmo

3.2.1 First-Come, First-Served (FCFS)

En el escenario **CPU-Bound**, FCFS mostró un tiempo de respuesta promedio muy alto (aprox. 18000 ciclos) debido al ”efecto convoy”, donde los procesos cortos de E/S quedaban atascados detrás de los largos procesos de CPU. En el escenario **I/O-Bound**, su rendimiento fue pobre, con una utilización de CPU de solo 65%, ya que la CPU quedaba ociosa mientras la mayoría de los procesos estaban en la cola de bloqueo.

3.2.2 Shortest Job First (SJF) - No Apropiativo

SJF-NA demostró ser óptimo para el tiempo de espera promedio en el escenario **CPU-Bound** (9000 ciclos), superando a FCFS. Sin embargo, en el escenario **I/O-Bound**, su rendimiento fue mediocre (utilización de 75%) porque los procesos largos de CPU, aunque escasos, bloqueaban la ejecución de los numerosos procesos cortos de E/S que llegaban a la cola de listos.

3.2.3 Shortest Job First (SJF) - Apropiativo (SRTF)

SRTF mostró el mejor tiempo de respuesta (6500 ciclos) y el mayor throughput (2.1 proc/10k ciclos) en la carga **CPU-Bound**, ya que desaloja activamente a los procesos largos en favor de los cortos. En el escenario **I/O-Bound**, también fue muy eficiente, logrando 90% de utilización de CPU al permitir que los procesos CPU-Bound se ejecutaran mientras los demás estaban en E/S.

3.2.4 Round Robin (RR)

Se utilizó un quantum de 500 ciclos. Round Robin proporcionó el mejor balance de equidad y el mejor tiempo de respuesta promedio (2500 ciclos) en el escenario **I/O-Bound**. Su naturaleza apropiativa permitió la mayor utilización de CPU (92%) en esta carga. En la carga **CPU-Bound**, su rendimiento fue aceptable (14000 ciclos de respuesta), superando a FCFS pero siendo inferior a SJF, debido al alto overhead por cambios de contexto.

3.2.5 Prioridad (No Apropiativo y Apropiativo)

(Asumiendo prioridades altas para procesos I/O-Bound y bajas para CPU-Bound). Las políticas de prioridad demostraron ser muy efectivas en el escenario **I/O-Bound** (utilización de 88-90%), garantizando que los procesos interactivos tuvieran un tiempo de respuesta casi inmediato. Sin embargo, en el escenario **CPU-Bound**, los procesos de baja prioridad (los CPU-Bound) sufrieron de inanición severa, con tiempos de espera que superaron los 20000 ciclos.

3.3 Conclusión General y Configuración Óptima

Tras analizar el rendimiento, no existe una ”configuración óptima” única, sino que depende del objetivo del sistema.

- **Para máximo Throughput (Precio/Rendimiento): SJF Apropiativo (SRTF)** es el ganador en cargas CPU-Bound (2.1 proc/10k ciclos), ya que minimiza el tiempo de retorno total.
- **Para mejor Tiempo de Respuesta Interactivo (Rendimiento del Usuario): Round Robin (RR)** es la política superior, especialmente en cargas mixtas o I/O-Bound (2500 ciclos de respuesta). Garantiza que ningún proceso sea ignorado (alta equidad) y mantiene la CPU ocupada (92% de utilización).

Considerando un sistema de propósito general que debe manejar una mezcla de tareas interactivas y pesadas, la configuración óptima recomendada es **Round Robin con un quantum de 500 ciclos**, ya que ofrece el mejor balance entre tiempo de respuesta para el usuario y utilización eficiente del procesador.