Simulador de Planificación de Procesos: MLFQ con RR y SJF

Proyecto SO-P1-2025-II

I. Introducción

Este proyecto implementa un **planificador multinivel con realimentación (MLFQ)**. El objetivo es simular la ejecución de procesos con diferentes políticas de planificación:

- **RR(1)**: Round Robin con *quantum* de 1 unidad de tiempo.
- **RR(3)**: Round Robin con *quantum* de 3.
- **RR(4)**: Round Robin con *quantum* de 4.
- **SJF**: Shortest Job First no expropiativo.

Cada proceso ingresa inicialmente en la primera cola y, cuando agota su *quantum* sin finalizar, desciende a la siguiente. Finalmente, si llega a la cuarta cola, se planifica mediante SJF.

II. Arquitectura de clases

- **Proceso**: encapsula la información de cada proceso (tiempos AT, BT, CT, WT, RT, TAT, estado, etc.).
- **RR**: implementa un planificador Round Robin con lista de listos y un iterador para recorrerla de forma circular.
- **SJF**: usa una `priority_queue` para ejecutar el proceso con menor *burst time*.
- **MLFQ**: contiene las tres colas RR y la cola SJF; decide en qué nivel ejecutar cada proceso y se encarga de moverlos de una cola a otra.
- **Simulador**: controla el tiempo global de la simulación, inyecta procesos cuando llega su *arrival time*, invoca al MLFQ y recopila métricas.
- **main.cpp**: punto de entrada, lee el archivo de entrada y lanza la simulación.

III. Flujo de ejecución

1. **Entrada**: `main.cpp` lee un archivo de texto con los procesos (label; BT; AT; Q; P).
2. Se crean instancias de `Proceso` y se ordenan por *arrival time*.
3. `Simulador` inicia un reloj de tiempo discreto. En cada tick:
   a) Inyecta los procesos cuyo *arrival* coincide con el tiempo actual en la primera cola (`RR(1)`).
   b) Llama a `MLFQ::run`, que:
   - Selecciona la cola más prioritaria no vacía.
   - Ejecuta el proceso en cabeza durante una unidad de tiempo.
   - Actualiza el tiempo de espera (`updWT`) en las demás colas.
   - Si el proceso finaliza, se elimina de la cola y se registran sus métricas.
   - Si agota su *quantum* sin terminar, se mueve a la cola de menor prioridad inmediata.
4. Al terminar todos los procesos, `Simulador` imprime un reporte tabular con CT, TAT, WT y RT y calcula sus promedios.

IV. Descripción de archivos y funciones

IV-A. *process.h / process.cpp*

- `class Proceso`: define los atributos de un proceso.
- `run(int time)`: ejecuta una unidad de CPU, reduce el *remaining*, registra *ResponseTime* la primera vez y, si *remaining* llega a 0, fija *CompletionTime*, *TurnAroundTime* y estado `TERMINATED`.
- `incrementWT()`: incrementa el tiempo de espera.

IV-B. *RR.h / rr.cpp*

- Implementa Round Robin con un `list<Proceso*>` y un iterador circular.
- `addProcess`: añade un proceso a la cola.
- `peek`: retorna el proceso apuntado por `current`.
- `increaseIT`: avanza el iterador circularmente.
- `deleteProcess`: borra el proceso actual y mueve el iterador.
- `updWT(int pid)`: incrementa el tiempo de espera de todos menos el que se está ejecutando.

IV-C. *SJF.h / SJF.cpp*

- Usa una `priority_queue` ordenada por *burst* restante, arrival y pid.
- `addProcess`: encola un proceso.
- `peek`: obtiene el de menor burst y lo saca de la cola.
- `updWT`: incrementa WT de todos los procesos.

IV-D. *MLFQ.h / MLFQ.cpp*

- Mantiene tres colas RR con distintos *quantum* y una cola SJF.
- `run(int time)`:
  - Si no hay proceso activo (`ac`), elige la cola más prioritaria con procesos.
  - Ejecuta una unidad de CPU, llama a `updWT` de cada cola.
  - Si el proceso termina, se elimina de la cola.
  - Si agota su *quantum*, `next()` lo mueve a la cola siguiente.
- `escogerCola`: determina el nivel de cola.

- `next`: inserta el proceso actual en la cola de menor prioridad inmediata.

IV-E. simulador.h / simulador.cpp

- Controla el tiempo global y la llegada de procesos.
- `run`: bucle principal: mientras haya procesos pendientes, incrementa el reloj, llama a `check()` para inyectar procesos nuevos y a `MLFQ::run`.
- `printReport`: imprime la tabla de métricas de cada proceso.
- `calculatePromedios`, `printPromedios`: calculan e imprimen promedios de CT, TAT, WT y RT.

IV-F. main.cpp

- Lee el archivo de entrada con formato `label;BT;AT;Q;P`, crea los procesos y lanza el `Simulador`.

V. Métricas de desempeño

Para cada proceso se calculan:

- **CT** (*Completion Time*)
- **TAT** (*Turn Around Time*) = CT - AT
- **WT** (*Waiting Time*)
- **RT** (*Response Time*) = primer instante en CPU - AT

El simulador imprime además los promedios de cada métrica.

VI. Análisis de casos de prueba

A continuación se presentan dos casos de prueba con su archivo de entrada y la salida producida por el simulador. El objetivo es mostrar cómo el planificador MLFQ con colas RR(1), RR(3), RR(4) y SJF gestiona procesos con distintas características.

VI-A. Caso de prueba 1

VI-A1. Entrada: -

Listing 1. Archivo de entrada

```
# etiqueta; burst time (BT); arrival time (AT);
    Queue (Q); Priority (5 > 1)
A; 8; 0; 1; 5
B; 9; 0; 1; 4
C; 10; 0; 2; 3
D; 15; 0; 2; 3
E; 6; 0; 3; 2
```

VI-A2. Salida generada: -

Listing 2. Reporte de ejecución

```
P   AT  BT   Q    CT   TAT   WT   RT
-------------------------------------
A   0   8    1    24   24    16   0
B   0   9    1    39   39    30   1
C   0   10   2    41   41    31   2
D   0   15   2    48   48    33   3
E   0   6    3    38   38    32   4
-----PROMEDIOS-----
Completion Time: 38
Turn Around Time: 38
Waiting Time: 28.4
Response Time: 2
```

VI-A3. Discusión de resultados (comportamiento real del código):

Aunque en el archivo de entrada cada proceso tiene una `Q` (cola) indicada, la implementación original **no** respeta ese campo: la función `MLFQ::addProcess` encola siempre en la primera cola (RR(1)), ignorando `getInitialQueue()`. Por eso, a pesar de que algunos procesos tienen `Q = 2` o `Q = 3`, en la ejecución observada todos comienzan en RR(1) y luego son degradados ("castigados") cuando agotan su quantum.

Este comportamiento explica la salida observada:

- Todos los procesos llegan en el tiempo 0 y son encolados inicialmente en RR(1).
- A, B, C, D y E agotan el quantum 1 y son degradados a la cola dos, todos esperaron la misma cantidad de tiempo, 4.
- Nuevamente todos consumen el quantum 3 enteramente y no terminan. Así pues nuevamente todos han esperado lo mismo 16.
- A y E terminan en la tercera cola, A consume el quantum entero, pero termina y E solo consume 2 de él. B, C y D son castigados a la siguiete cola.
- En la cuarta y última cola los procesos terminan, primero A, pues le su Job era menor con 1, luego B con un BT restante de 2 y por último C con un BT restante de 7.

VI-B. Caso de prueba 2

VI-B1. Entrada: -

Listing 3. Archivo de entrada

```
p1; 20; 0; 1; 5
p2; 10; 2; 1; 4
p3; 15; 4; 2; 3
p4; 5; 6; 3; 2
p5; 8; 8; 3; 1
```

VI-B2. Salida generada: -

Listing 4. Reporte de ejecución

```
P   AT   BT   Q   CT   TAT   WT   RT
-------------------------------------
p1  0    20   1   58   58    38   0
p2  2    10   1   39   37    27   4
p3  4    15   2   46   42    27   5
p4  6     5   3   33   27    22   6
p5  8     8   3   37   29    21   10
-----PROMEDIOS-----
Completion Time: 42.6
Turn Around Time: 38.6
Waiting Time: 27
Response Time: 5
```

VI-B3. Discusión de resultados (comportamiento real del código):

Al igual que en el primer caso, la función `MLFQ::addProcess` siempre encola los procesos en la **primera cola** (RR(1)), ignorando el campo `Q` especificado en la entrada. Por ello todos los procesos inician en RR(1) y sólo descienden cuando agotan su *quantum*, siguiendo el mecanismo de "castigo" del MLFQ.

- **Llegadas escalonadas:** los procesos no arriban todos en el mismo instante. p1 llega en $t = 0$, p2 en $t = 2$, p3 en $t = 4$, p4 en $t = 6$ y p5 en $t = 8$. Esto produce diferencias marcadas en los tiempos de respuesta: p1 obtiene CPU de inmediato (RT=0) mientras p5 debe esperar hasta que se libere espacio (RT=10).
- cada proceso, al llegar, ejecuta en la primera cola con *quantum* 1. Ninguno completa su *burst* allí, de modo que todos van siendo degradados gradualmente. P1 consume su Quantum y aún no ha llegado nadie mas, así que inmediantamente pasa a ejecución desde la cola 2. También consume el q 4 enteramente, P2, P3 Y P4 usan su quantum vaciando la cola 1 en el tiempo 7, no ha llegado P5 así que sigue con P2 en la cola 2, para consumir ahora un quantum de 3. Entonces P5 se ejecuta desde la primera cola, por eso es atendido en el tiempo 10.
- Continuan P3, P4 y P5 desde la cola 2, consumiendo el quantum de 3. Ninguno termina aún.
- **Fase RR(4) y SJF:** p4 y p5 terminan, p1 consumiendo solo 1 y P3 consumiendo los 4 enteros del quantum en esta tercera cola. P1, P2, P3 y P4 consumen el quatum entero y no terminan, pasan a la ultima cola. Primero termina P2 con el trabajo mas corto pues le restaban unicamente 2 de BT, continua P3 con 7 restantes y por último P1 con 12. .

Este segundo escenario ilustra el **efecto combinado de la política MLFQ y las llegadas en distintos instantes**: los procesos tempranos (p1, p2) obtienen CPU rápidamente, mientras que los tardíos (p4, p5) ven aumentado su tiempo de respuesta a pesar de tener ráfagas cortas.

VII. Conclusiones

El esquema MLFQ con niveles RR(1), RR(3), RR(4) y SJF permite balancear la equidad inicial (con *quantum* muy corto) y la eficiencia en procesos largos, degradando gradualmente los procesos que requieren más CPU hacia niveles de menor prioridad. La implementación modular facilita reemplazar políticas y ajustar parámetros como el quantum.