**SCHEDULING STRATEGIES IN OPERATING SYSTEMS: IMPLEMENTATION AND BENCHMARKING OF KEY ALGORITHMS IN JAVA**

**ESCUELA DE INGENIERÍA CIENCIA Y TECNOLOGÍA**

**UNIVERSIDAD DEL ROSARIO**

Sebastian Dario Perez Pantoja, David Hernando Sánchez Lombana

e-mail: [sebastiand.perez@urosario.edu.co](mailto:sebastiand.perez@urosario.edu.co) , [davidh.sanchez@urosario.edu.co](mailto:davidh.sanchez@urosario.edu.co)

**RESUMEN**

El reporte se centra en la implementación de métodos fundamentales de scheduling dentro del simulador UR-OS una application basada en Java, que intenta incluir los elementos más importantes de un sistema computacional.

PALABRAS CLAVE: Process Scheduling, Java, Operating Systems, Computational Complexity.

1. **INTRODUCCIÓN**

La planificación de procesos, o process scheduling, es un componente esencial en los sistemas operativos modernos, ya que gestiona la asignación de la CPU entre múltiples procesos en ejecución. Su correcto funcionamiento impacta directamente en el rendimiento del sistema, optimizando el uso de recursos y mejorando la experiencia del usuario.

En este reporte, se presentan e implementan cinco algoritmos de planificación: First-Come, First-Served (FCFS), Shortest Job First No Apropiativo (SJF NP), Shortest Job First Apropiativo (SJF P), Round Robin (RR) y Multilevel Feedback Queue (MFQ). La implementación se lleva a cabo en la versión 0.0.3.8 del sistema UR OS, desarrollado por Pedro Wightman para el curso de Sistemas Operativos.

El objetivo de este trabajo es comprender cómo interactúan la CPU y el sistema operativo en la gestión de múltiples procesos, analizando la eficiencia y el impacto de cada algoritmo en el rendimiento del sistema.

1. **FUNDAMENTOS DEL SCHEDULING DE PROCESOS: DEFINICION, TIPOS DE SCHEDULING, METRICAS CLAVES.**

Los métodos de ordenación de procesos, se basan en algoritmos diseñados para optimizar el uso de la CPU y mejorar el rendimiento del sistema, medible por parte de una serie de métricas que mencionaremos proximamente.

Referente a los algoritmos que se probarán en para este reporte, tenemos:

* + First-Come, First-Served (FCFS): Asignación de procesos en orden de llegada sobre la cola de ejecución o Ready Queue.
    - Shortest Job First Non Preemptive (SJF NP): Asignación de procesos con prioridad sobre aquellos con menor tiempo de ejecución estimado. No permite interrupciones una vez el proceso haya llegado al CPU.
    - Shortest Job First Preemptive (SJF P): Asignación de procesos con prioridad sobre aquellos con menor tiempo de ejecución estimado. En caso de que aparezca en la ReadyQueue un proceso con menor tiempo de ejecución, se interrumpe la cpu y se re-evalua en busqueda del proceso con menor tiempo de ejecución estimado.
    - Round Robin (RR): Se define un tiempo fijo en terminos de ciclos bajo el nombre de quantum. Cada vez que termine el ciclo del quantum, se interrumpe la CPU y el proceso es enviado al final de la cola.
    - Multilevel Feedback Queue (MFQ): Se basa en la utilización de multiples colas, cada una con un nivel de prioridad diferente. Se priorizan los procesos con menor valor de este parametro.

Referente a las métricas clave para evaluar el rendimiento de estos procesos tenemos:

* + Waiting Time: Tiempo total que un proceso pasa en la cola de espera antes y entre sus ejecuciones.
  + Turnaround Time: Tiempo total desde que un proceso llega al sistema hasta que finaliza su ejecución.
  + Response Time: Tiempo desde que un proceso llega al sistema hasta que se ejecuta por primera vez.
  + Throughtput: Número de procesos completados por unidad de tiempo
  + CPU Utilización: Porcentaje de tiempo de trabajo de la CPU a lo largo del periodo de procesamiento.

1. **IMPLEMENTACIÓN: FCFS, SJN NP, SJN P, RR.**

A continuación se presenta la implementación en pseudo-codigo de los diferentes métodos de scheduling trabajados.

**Casos Apropiativos:**

Dentro de cada uno de los métodos de scheduling, existe un método getNext(boolean cpuEmpty), encargado de que se cumpla el parametro de ejecución.

Para los casos *Non Preemptive* las funciones newProcess y IOReturningProcess existen dentro de las clases referentes a los schedulings, sin embargo, no son utilizados.

Adicionalmente, es importante enunciar la importancia de la funcion *interrupt* dentro de la clase *Os*, pues es el medio por el que insertamos un proceso dentro del CPU. A su vez, sirve para extraerlo, cuestión útil a la hora de usar métodos *Preemptive.*

* **FCFS:**

Mientras haya procesos en el sistema:

Si un nuevo proceso llega a RQ Entonces:

Agregar el proceso al final de RQ // Se mantiene el orden de llegada

Si la CPU está libre y RQ no está vacía Entonces:

Seleccionar el primer proceso en RQ

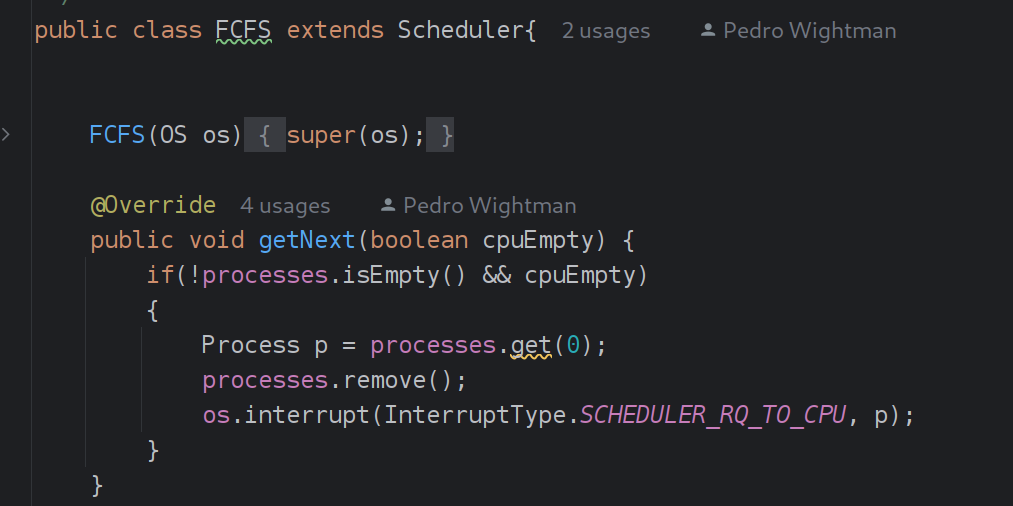
Remover proceso seleccionado de RQ

Asignar proceso a la CPU

Ejecutar hasta su finalización // No hay interrupciones

Si el proceso en CPU termina su ejecución Entonces:

Marcar proceso como completado



En otras palabras para el FCFS o FIFO, lo que buscamos es evaluar en los casos en los que el CPU esté vacío (método cpuEmpty), si la lista de procesos está o no vacía. En caso de que existan elementos dentro de esta, se extrae aquel ubicado en el primer indice de la estructura, y se interrumpe el CPU para ejecutarlo hasta su finalización.

* **SJN NP:**

Definir min\_BTR = Max\_sim\_cycles // Valor máximo inicial para comparar

Definir min\_BTR\_process = NULL // Variable para almacenar el proceso con menor Burst Time restante

Para cada proceso p en la lista de procesos:

temp ← BurstRemainingTime(p) // Obtener tiempo de ráfaga restante del proceso

Si temp < min\_BTR Entonces

min\_BTR ← temp

min\_BTR\_process ← p

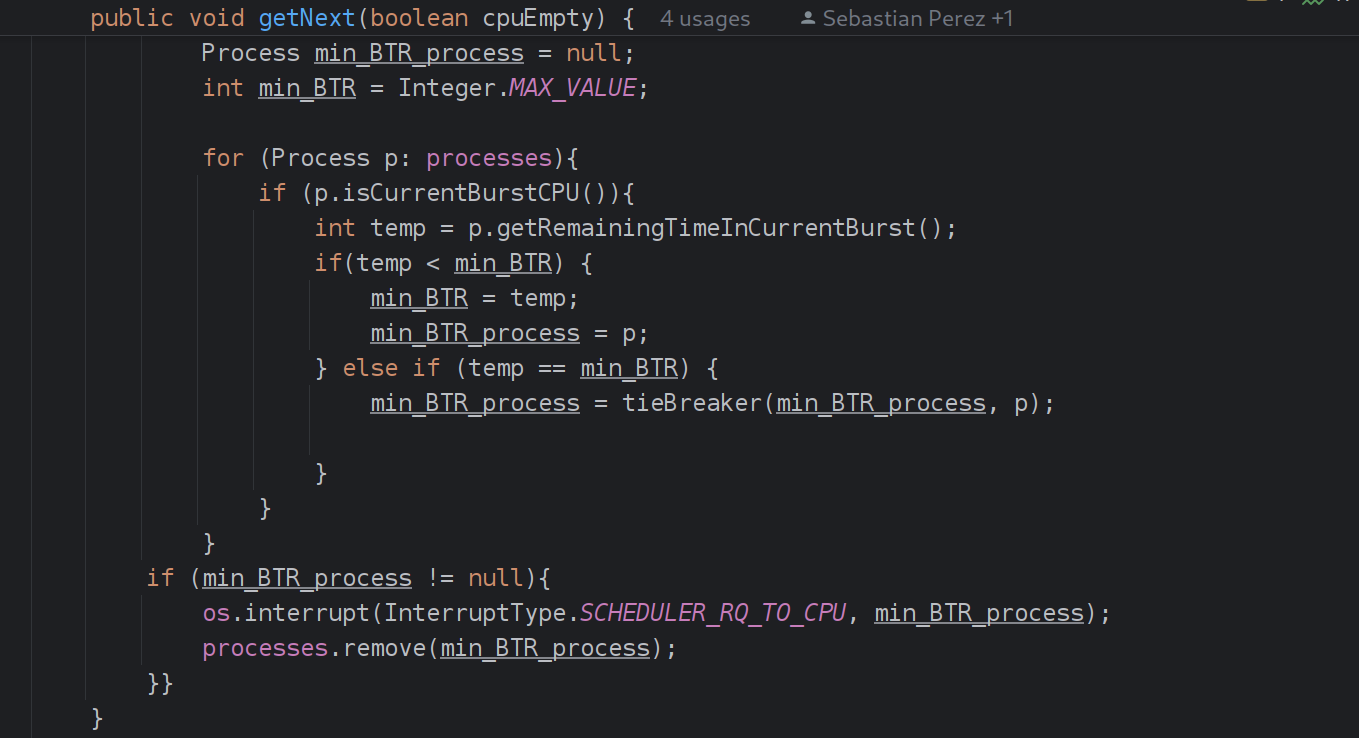
Sino Si temp = min\_BTR Entonces

Aplicar criterio de desempate (Tie-Breaker)

Si min\_BTR\_process ≠ NULL Entonces

Eliminar min\_BTR\_process de la lista de procesos

Llamar a InterruptHandler(RQ\_TO\_CPU, min\_BTR\_process) // Asignar proceso a la CPU



Para esta implementación, el getNext() rastrea el proceso con menor cantidad de ciclos restantes de CPU. Esto se consigue mediante la iteración de cada uno, y la comparación del parametro RemainingTimeInCurrentBurst.

En caso de que dos procesos tengan el tiempo mínimo, se utiliza el parametro definido en TieBreaker.

* **Round Robin:**

Mientras haya procesos en el sistema:

Si un nuevo proceso llega a RQ o un proceso sale de I/O a RQ Entonces:

Si hay un proceso ejecutándose en la CPU Entonces:

Interrumpir proceso en CPU

Enviar proceso interrumpido de vuelta a RQ

Si cont == q Entonces // Si el quantum del proceso en CPU se ha agotado

Resetear contador cont = 0

Si hay un proceso en la CPU Entonces:

Interrumpir proceso en CPU

Enviar proceso interrumpido de vuelta a RQ

Llamar a getNext(true) // Buscar siguiente proceso en cola

Sino // Si aún no se ha agotado el quantum, continuar ejecutando

Incrementar cont en 1

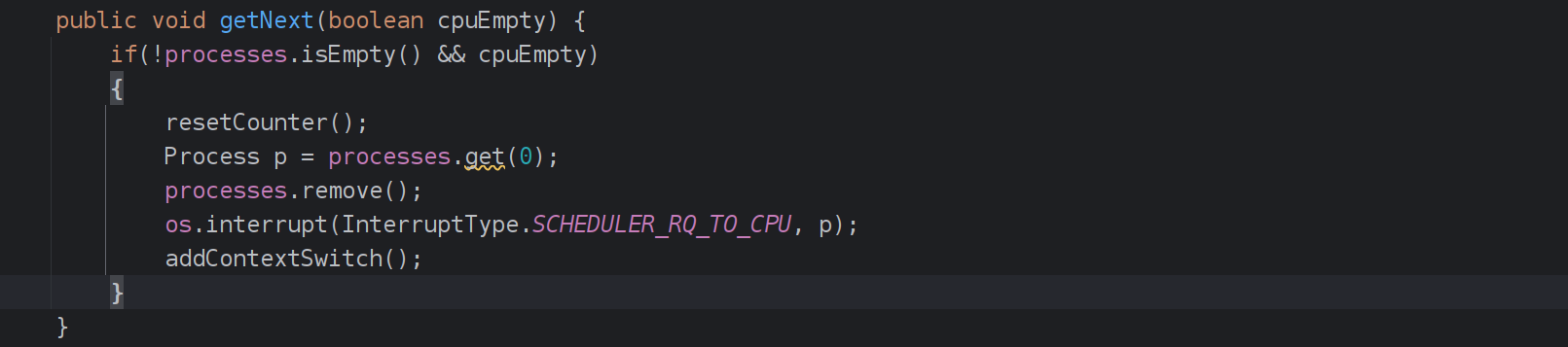
Si la CPU está vacía y hay procesos en RQ Entonces:

Seleccionar el primer proceso en la lista de procesos (FIFO)

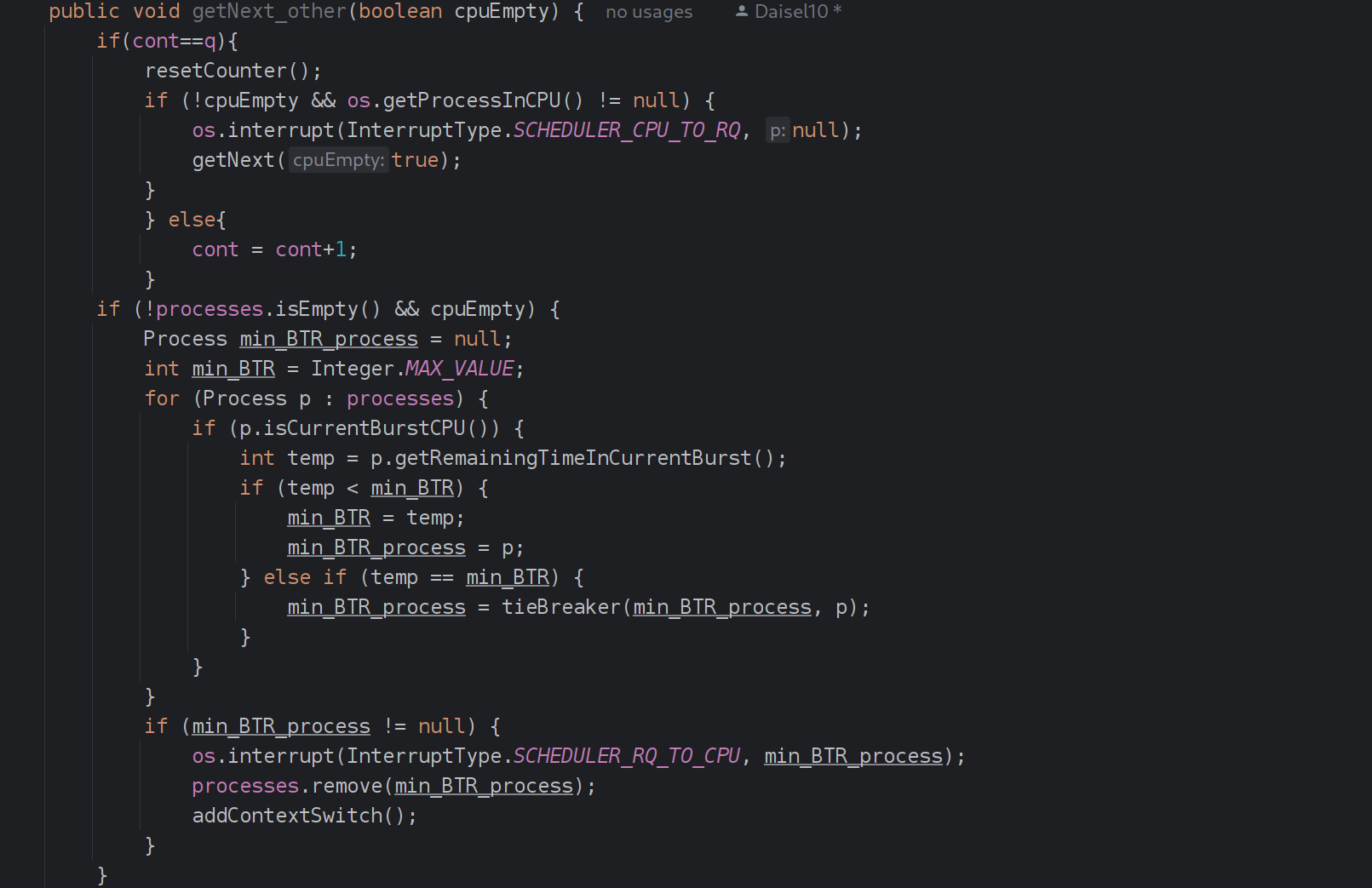
Eliminarlo de la lista de procesos

Llamar a InterruptHandler(RQ\_TO\_CPU, proceso) // Asignar proceso a la CPU

Resetear contador cont = 0 // Iniciar quantum nuevamente para este proceso



Nuestro getNext funciona como un FIFO. Si la ejecución llega a él antes de que el contador cumpla el ciclo definido por el valor del quantom, se ejecuta el proceso siguiente mientras se reinicia el contador.



En el caso en el que el contador llega al valor del quantum, se reinicia este y se interrumpe el proceso, sacandolo de la CPU e insertandolo en el ReadyQueue. Una vez ocurra esto, Round Robin aplica el método base del SJF para extraer el nuevo método a insertar en CPU.

**No Apropiativos:**

Para los procesos Preemptives, utilizamos las funciones void newProcess y IOReturning, pues son momentos de la simulación donde aparece un nuevo candidato a evaluar. Aquí lo que hacemos es por medio de la función interrupt, remover el proceso de la CPU para insertarlo de nuevo en la lista de Procesos.

* SJF P:

Mientras haya procesos en el sistema:

Si un nuevo proceso llega a RQ o un proceso sale de I/O a RQ Entonces:

Si hay un proceso ejecutándose en la CPU Entonces:

Interrumpir proceso en CPU

Enviar proceso interrumpido de vuelta a RQ

Definir min\_BTR = Max\_sim\_cycles // Valor máximo inicial para comparar

Definir min\_BTR\_process = NULL // Variable para almacenar el proceso con menor Burst Time restante

Para cada proceso p en la lista de procesos:

temp ← BurstRemainingTime(p) // Obtener tiempo de ráfaga restante del proceso

Si temp < min\_BTR Entonces

min\_BTR ← temp

min\_BTR\_process ← p

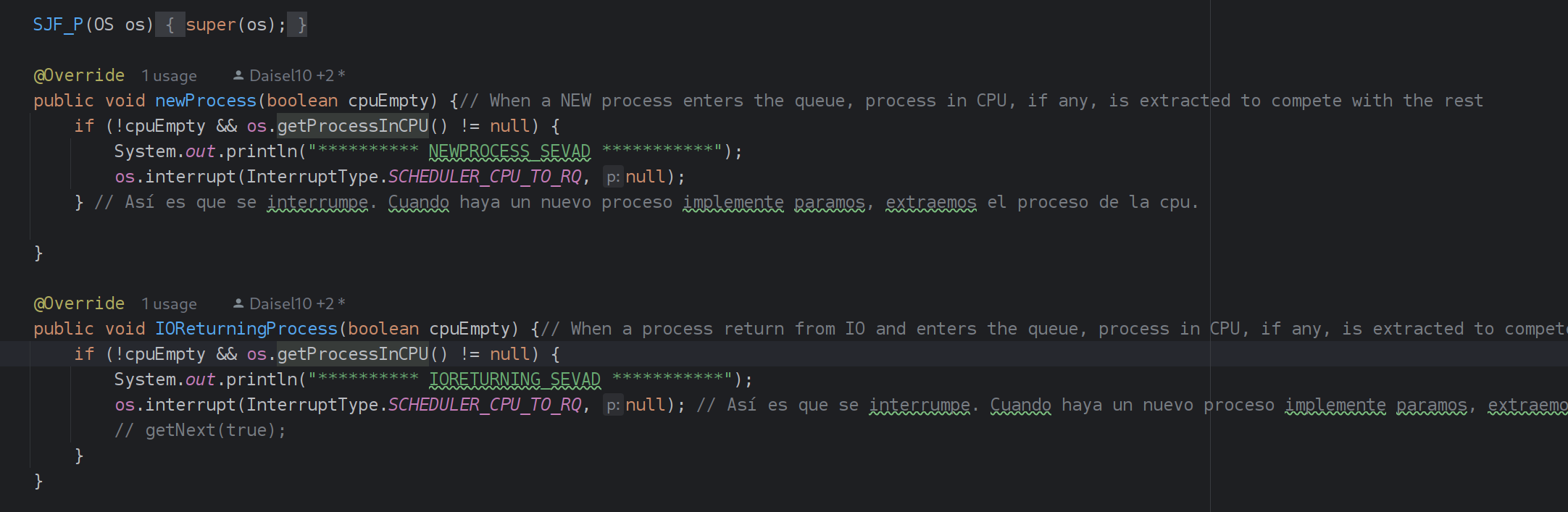
Sino Si temp = min\_BTR Entonces

Aplicar criterio de desempate (Tie-Breaker)

Si min\_BTR\_process ≠ NULL Entonces

Eliminar min\_BTR\_process de la lista de procesos

Llamar a InterruptHandler(RQ\_TO\_CPU, min\_BTR\_process) // Asignar proceso a la CPU



Se implementa la misma lógica que en la versión SJF NP. El cambio cómo se mencionó anteriormente es sobre las funciones newProcess y IOReturning por las razones anteriormente mencionadas. Adicionalmente, se colocan condicionales para evitar que se extraigan procesos en momentos donde no los hay; lo que carecería de sentido.

1. **PRUEBAS Y BENCHMARKING.**

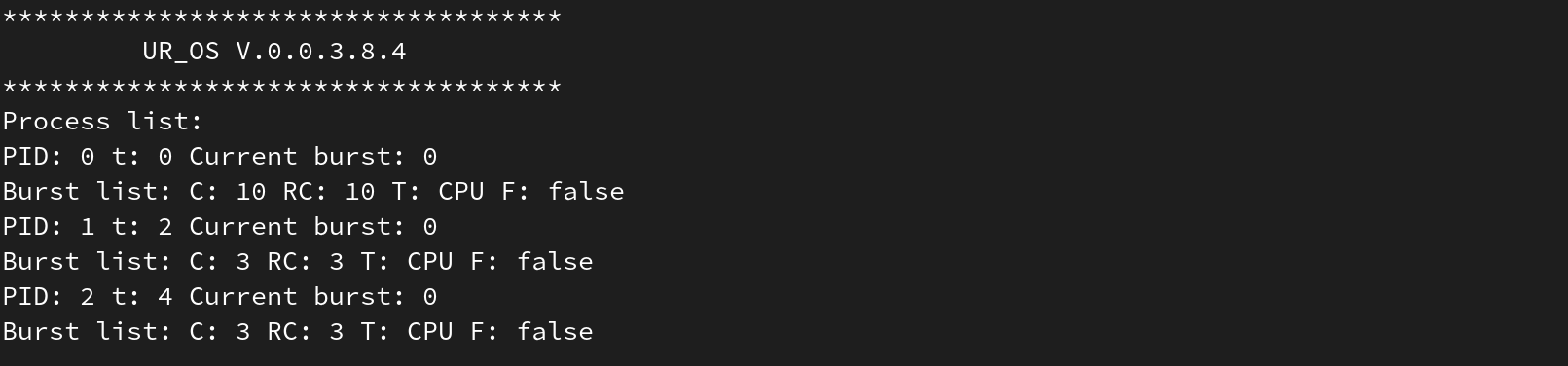
Para las pruebas, hemos decidido crear nuestras propias simulaciones, enfocadas en medir el correcto funcionamiento de los parámetros de cada método de scheduling.

El TieBreaker elegido por el equipo fue SMALLEST\_PID, pues nos pareció más congruente que usar el más alto.

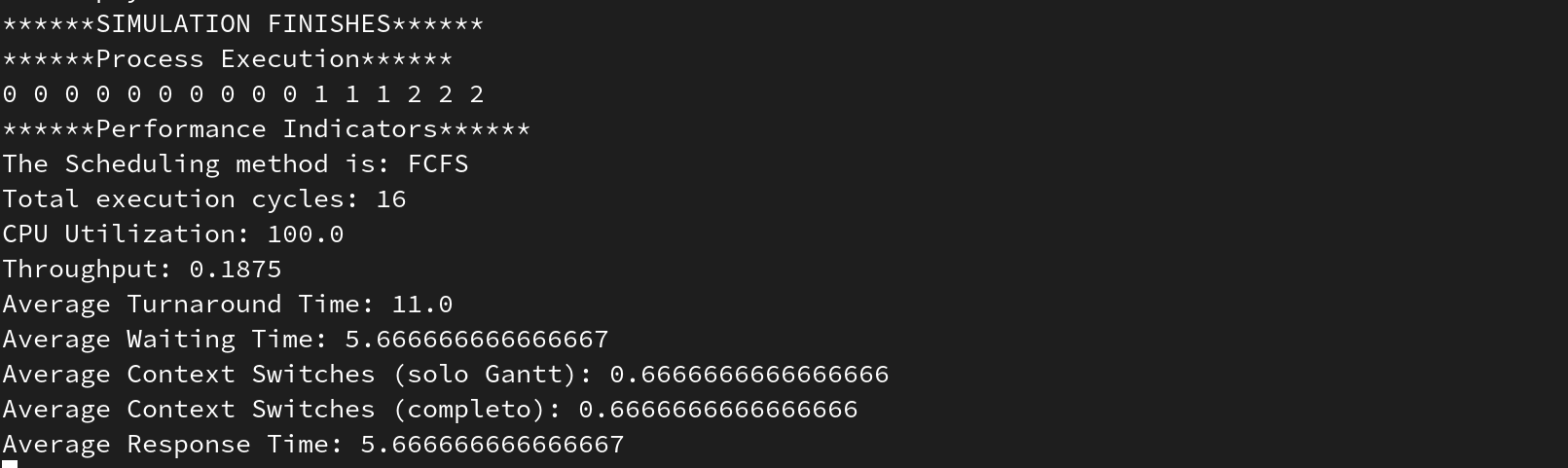
* **FCFS:**

Para el primer caso, hemos definido una simulación con un total de 3 procesos. Estos procesos tienen una estructura básica, donde solo hay un elemento en el BurstList, ciclos de CPU.

Los procesos contenidos en la simulación (simulacion\_FCFS0()) son los siguientes:

****

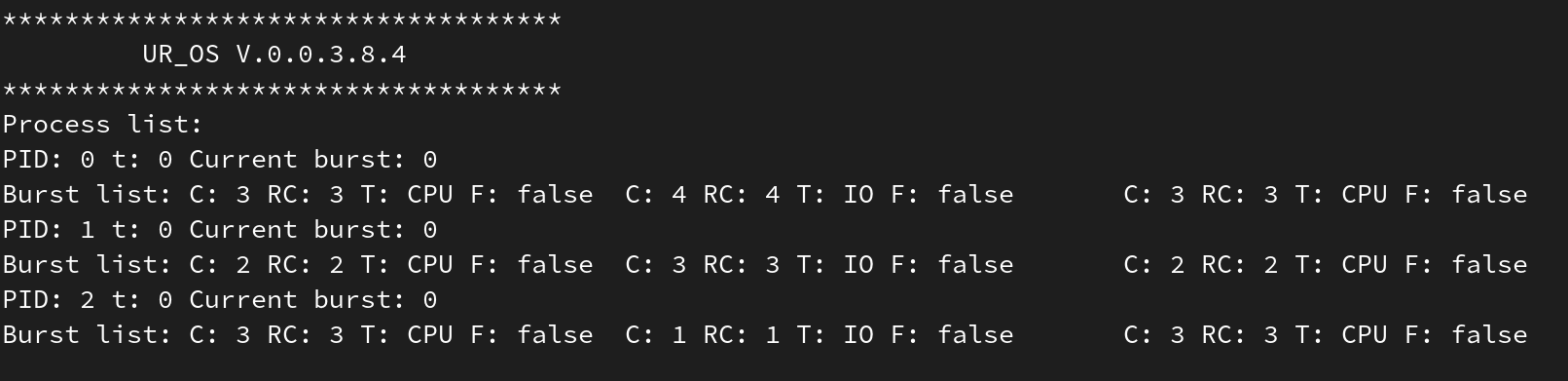
Y el proceso de ejecución junto a sus métricas es el siguiente:

****

Aquí podemos ver, como se cumple el parametro de FIFO, el primero que entra, el primero que sale.

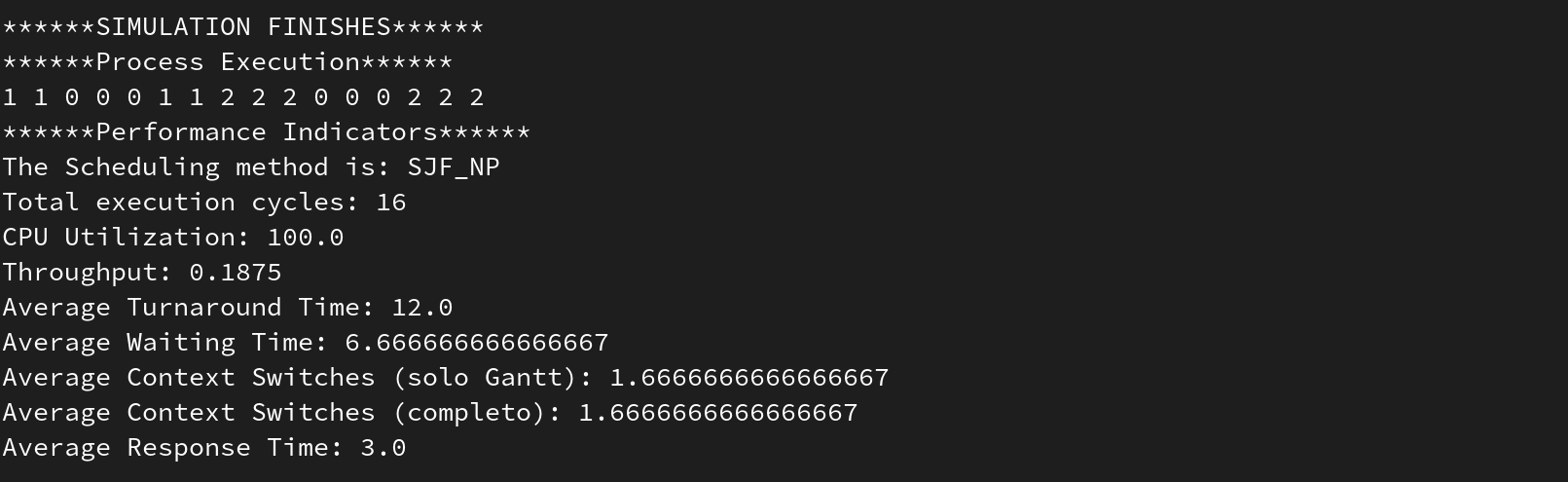
SJF NP:

Para este caso, nuestros procesos además tienen ciclos en I/O, lo que nos permite aplicar nuestro algoritmo de comparación sobre los procesos.

****

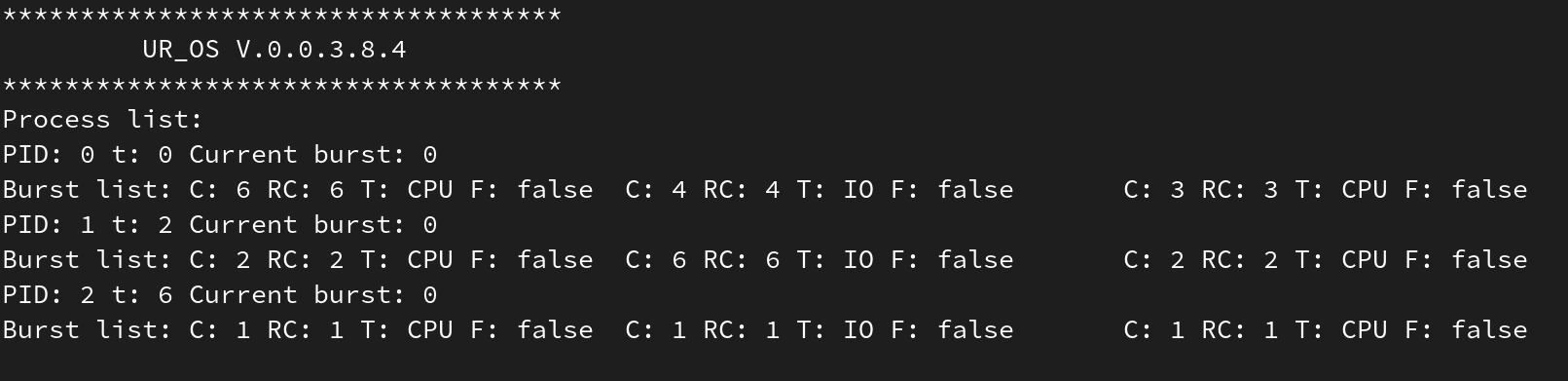
Podemos ver que al principio de la ejecución, se ejecuta dentro del getNext() la comparación, insertando el proceso con PID 1 al CPU. Luego de este vaya a I/O, el TieBreaker define que PID: 0 continue. Cuando este termina re evalua y encuentra en PID 1 el candidato ideal al tener 2 ciclos, un ciclo menos que PID 2. La ejecución continua con esta lógica hasta finalizar.

Podemos ver a continuación el proceso de ejecución junto con las métricas de desempeño:

****

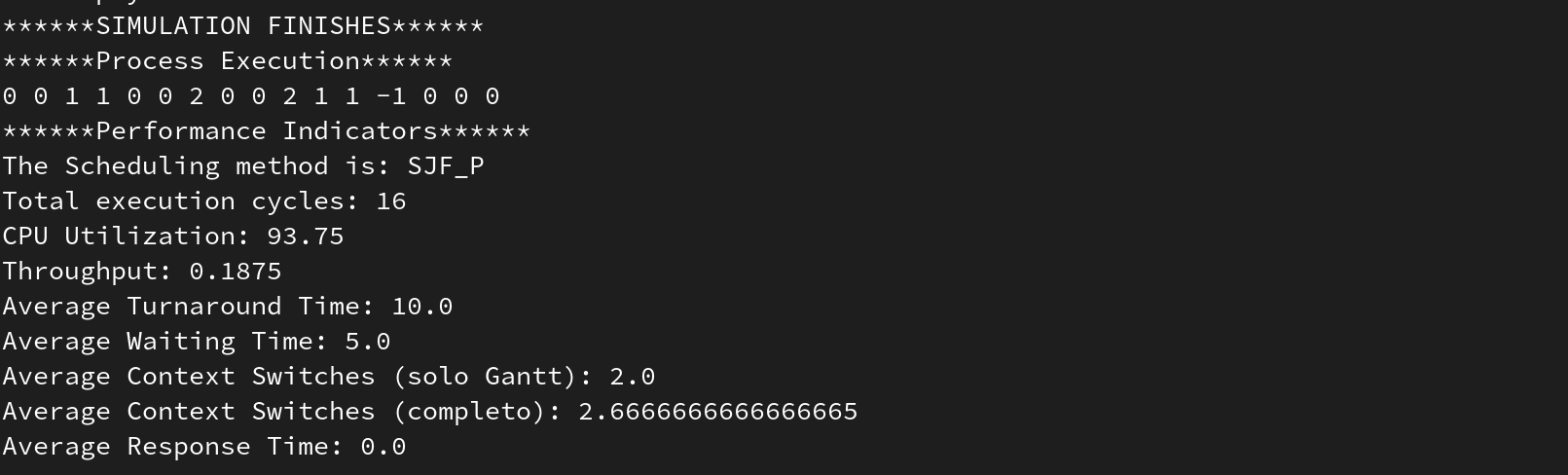
SJF P:

Para la versión Preemptive del SJF, creamos una simulación en el que arranca primero y último el proceso con ciclos más largos, y el que contiene los ciclos más cortos respectivamente.

****

En este caso, lo que buscamos fue que hubieran interrupciones constantes debido al parametro de ciclo más corto, invocado por newProcess y IORETURNING.

Esto se puede ver a continuación, junto con las métricas de desempeño:

****

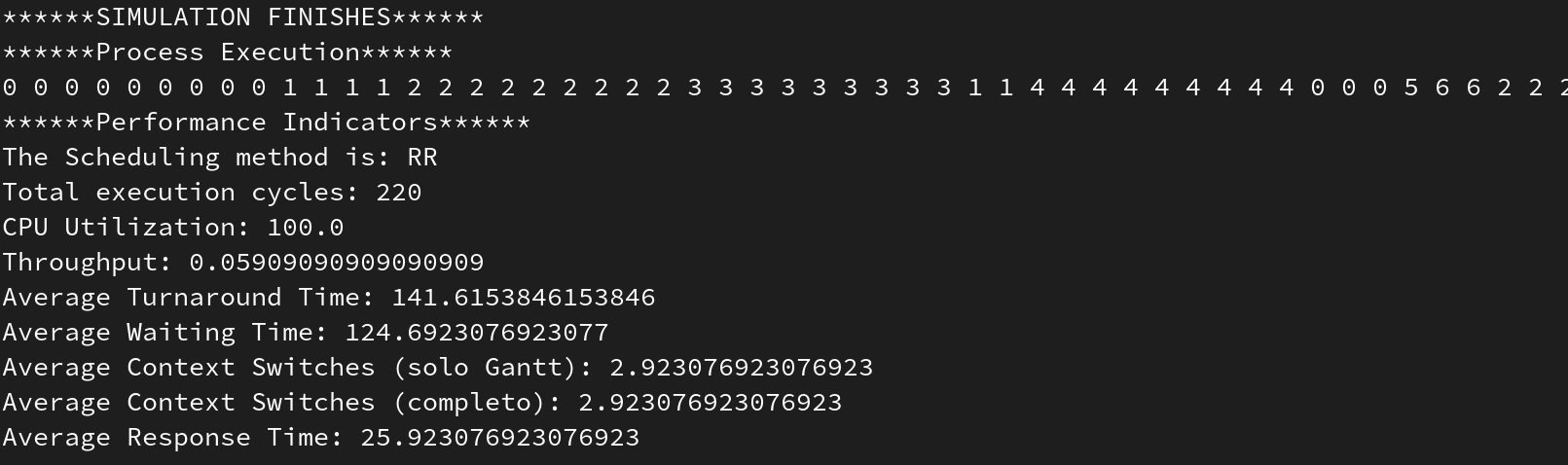
Round Robin:

Para Round Robin, preferimos tomar la simulación initSimulationQueueSimple.

A continuación podemos ver los procesos asociados a esta simulación:



Los resultados de la simulación son los siguientes:

****

1. **DESAFÍOS Y CONSIDERACIONES.**

Entre los desafíos que encontramos, está el primero de ellos el entendimiento de cómo funcionan los métodos de scheduling, pues a pesar de que son gestionados por el Sistema Operativo, son atravesados por un sin fin de funciones y métodos de este y de la CPU.

Adicionalmente, quedan algunas dudas sobre cómo responden diversos parametros del sistema operativo durante la ejecución, como el número máximo de procesos en TieBreaker.

Por último, la posibilidad de medir el nivel de funcionamiento de los métodos a partir de las simulaciones alojadas en el Sistema Operativo, pues son extensas y generales. Encontramos como una mejor salida, el planteamiento de simulaciones cortas, enfocadas en los métodos para analizar el correcto funcionamiento del Sistema Operativo.

1. **CONCLUSIONES**

Los métodos de scheduling, a pesar de lo que puede parecer en términos de eficiencia, tienen diversos casos de uso, permitiendo incluso que algoritmos como FCFS sigan siendo relevantes en ciertos contextos prácticos, y no solo como un modelo teórico o caso base.

El simulador está estructurado correctamente y permite visualizar de manera clara el comportamiento del CPU y del sistema operativo en la ejecución de procesos, facilitando la comprensión de los algoritmos de planificación.

La implementación de un método de interrupciones para manejar eventos como la llegada de nuevos procesos o la ejecución de uno refleja con precisión el funcionamiento de los sistemas operativos modernos.