# *PRÁCTICA 1: PLANIFICACIÓN DE PROCESOS*

## DISEÑO DE SISTEMAS OPERATIVOS

**Curso 2015-2016**


**Grado en Ingeniería Informática**

**Universidad Carlos III de Madrid**


| | |
|---|---|
| **JAVIER PRIETO CEPEDA:** | **100 307 011** |
| **SERGIO RUIZ JIMÉNEZ:** | **100 303 582** |
| **MARIN LUCIAN PRIALA:** | **100 303 625** |

# ÍNDICE



# TABLAS DE PRUEBAS

# 1 Introducción

En esta memoria se tratarán los aspectos más relevantes de la práctica número uno de la asignatura Diseño de Sistemas Operativos.

Para ello se comenzará describiendo brevemente el código implementado para las distintas políticas de planificación. Cabe destacar que son tres políticas distintas y las dos últimas parten de la base de la primera política.

A continuación, se compararán cada una de las políticas de planificación, obteniendo las ventajas y desventajas de cada una de ellas.

En último lugar se comentarán las pruebas llevadas a cabo y los resultados de cada una de ellas, incluyendo al final del documento una breve conclusión de la práctica realizada.

En el presente documento no se comentará el rendimiento de cada planificador, ya que todos tardarán el mismo tiempo en ejecutar todos los procesos debido a que la implementación es para tan solo un núcleo.

# 2 Descripción del código

## 2.1 Round-Robin

Round-Robin es una política de planificación que permite ejecutar los procesos de una manera equitativa, debido a que cada proceso tiene una rodaja de tiempo para su ejecución. Esta rodaja suele ser denominada quantum o time-slot.

Para su implementación se han debido añadir dos campos nuevos a cada proceso a la hora de crear cada proceso, que han sido el tiempo de la rodaja y su id. Cabe destacar también la necesidad de crear una cola donde almacenar los procesos listos para su ejecución.

Para la implementación de esta política se han usado las tres funciones principales timer_interrupt(), scheduler() y activator().

Y adicionalmente se ha modificado la función de finalización del proceso/hilo llamada: mythread_exit().

A continuación, se detallarán las funciones anteriormente citadas:

***Timer_interrupt():*** Cada vez que se accede a esta función de debe disminuir el tiempo de rodaja del proceso, ya que lo consume.

Se debe tener en cuenta cuando la rodaja de tiempo se ha acabado, si el proceso no ha acabado su tiempo de ejecución o si por lo contrario se ha acabado y debe ejecutar el siguiente proceso. Todo ello se debe hacer con acceso protegido para evitar problemas al llamar al siguiente proceso. Esta última acción se repetirá siempre que queramos obtener un nuevo proceso.

***Scheduler():*** La función deberá devolvernos el siguiente proceso a ejecutar, es decir, se encarga de determinar qué proceso debe ejecutar. Para ello hay que comprobar si la cola de procesos está vacía o no, en caso de no estar vacía significa que hay procesos que aún deben ejecutar, obteniendo el primero de la cola. Si no existiera ningún proceso en la cola de listos, y el proceso actual no hubiera finalizado su ejecución, pero si su rodaja, volverá a ejecutarse.

Si no hubiera procesos en la cola y el proceso actual hubiera finalizado, el programa puede finalizar.

***Activator():*** Es la función que se encarga de poner a ejecutar el siguiente proceso, ya sea por finalización de la rodaja o del propio proceso. Por ello tiene un parámetro, que es el proceso que deberá poner a ejecutar, previamente determinado por el scheduler.

Una vez el scheduler ha determinado el siguiente proceso, el activator se encarga de comprobar si el proceso ha acabado o si por el contrario está listo. En caso de estar finalizado, se pone a ejecutar otro proceso. Cuando un proceso ha finalizado usa la función mythread_exit().

Si el proceso aún tiene tiempo de ejecución y por lo tanto está listo, deberá realizar un swap entre procesos, guardando el estado del anterior y metiéndolo de nuevo en la cola por no haber finalizado. Y realiza el cambio de contexto entre ellos.

***Mythread_exit():*** Es la función que se encarga de finalizar un proceso y liberar sus recursos. También se le añade la funcionalidad de imprimir que el proceso ha terminado su ejecución no por fin de rodaja, comprobando si el proceso debe seguir ejecutando porque es el único disponible o si ha finalizado su ejecución. Imprimiendo su id y el nuevo id del proceso siguiente.

## 2.2 Round-Robin/FIFO con prioridades

Este nuevo planificador introduce un cambio significativo al anterior. Los procesos tienen distintas prioridades: baja o alta. Por lo tanto, se ejecutarán de la siguiente manera:

1. Si el proceso tiene una prioridad alta deberá ser introducido en una cola donde se encuentran los procesos de prioridad alta, que serán ejecutados en modo FIFO.
2. Si el proceso tiene prioridad baja deberá estar en una cola donde solo existan procesos de baja prioridad, que serán ejecutadas en modo Round-Robin.
3. Si el proceso que está en ejecución es de prioridad baja y existen procesos en la cola de prioridad alta, deberán parar su ejecución modificando su variable de tiempo de ejecución al valor inicial y poner a ejecutar todos los procesos de la cola de alta prioridad.

Para ello se han usado las funciones anteriormente descritas para Round Robin y se ha modificado la función de creación de hilos para introducir cada proceso en su correspondiente cola, dependiendo de su prioridad y en caso de existir procesos de alta prioridad mientras ejecuta uno de baja prioridad expulsarlo.

***Timer_interrupt():*** Partiendo de la base de Round-Robin, se añade una pequeña modificación, ya que no solo se genera una interrupción en los procesos por finalización de rodaja sino también por ser un proceso de prioridad baja. Esto es debido a que si el proceso tiene prioridad baja puede ser interrumpido por otro proceso de prioridad alta (Esta interrupción es tratada en la función mythread_create().)

***Scheduler():*** La función se encarga ahora de determinar el proceso que debe ser ejecutado en función de su prioridad. Para ello primero debe comprobar si existe algún proceso en la cola de alta prioridad y devolverlo, o por el contrario no quedan procesos de alta prioridad y debe obtener un proceso de baja prioridad.

Como anteriormente se explicó, en caso de no haber procesos en ninguna cola y el actual no estar finalizado, deberá ser retornado para que acabe su ejecución.

***Activator():*** Se añade una nueva funcionalidad, ya que cuando se realiza un cambio de contexto se debe terminar la prioridad del proceso para meterlo en la cola correspondiente guardando en su estado.

## 2.3 Round-Robin/FIFO con starvation free

El siguiente planificador es el más costoso ya que se quiere evitar en modo FIFO el starvation free, es decir, que todo proceso se ejecute, ya que en algunos casos

podría ocurrir que un proceso siempre quisiera ejecutar, pero al existir procesos de prioridad alta no pudiera, quedándose siempre preparado.

Por ello es necesario añadir una pequeña modificación, recorriendo la lista de procesos de baja prioridad para comprobar e disminuir su variable de hungry previamente inicializada al valor de starvation. Si un proceso de la cola de baja prioridad llega a 0, deberá ser pasado a proceso de alta prioridad metiéndolo en la correspondiente cola.

Este cambio introducido se realiza en la función timer_interrupt().

***Timer_interrupt():*** Se añade la funcionalidad de recorrer la cola de procesos de baja prioridad siempre que no esté vacía, disminuyendo su variable hungry hasta 0 por cada llamada al método. En caso de alcanzar 0 en algún momento, el proceso que haya alcanzado ese valor debe reiniciar su valor hungry al propuesto por defecto y ser introducido como proceso de alta prioridad en la correspondiente cola, previamente extrayéndolo de la cola de baja prioridad.

Todas las demás funciones son similares, ya que sigue los mismos patrones con está adición para evitar la hambruna o starvation.

# 3 Ventajas y desventajas

Previamente se han descrito las distintas políticas de planificación y su implementación de una manera objetiva.

En este apartado se detallarán sus posibles ventajas y desventajas.

## 3.1 Round-Robin

Ventajas:

a. No existe starvation free.
b. Es una planificación equitativa, ya que todos los procesos tienen la misma rodaja de tiempo.
c. Es fácil de implementar y escalable.

Desventajas:

a. Al ser una planificación equitativa los procesos de mayor prioridad no tienen por qué ser ejecutados primero.
b. No se premia a los procesos de mayor duración.
c. Es escalable pero no es deseable su implementación por las razones descritas anteriormente.

## 3.2 Round-Robin/FIFO con prioridades

Ventajas:

a. Es una planificación que premia a los procesos de alta prioridad para que sean finalizados primeros.
b. Es fácil de implementar y escalable.

Desventajas:

a. Existe starvation, ya que algunos procesos de prioridad baja pueden no ser ejecutados.
b. Aunque es escalable el sistema, al existir starvation con algunos procesos, no es recomendable si existen muchos procesos de prioridad baja, ya que podrían no ser ejecutados.

## 3.3 Round-Robin/FIFO con starvation free

Ventajas:

a. Es una planificación que premia a los procesos de alta prioridad para que sean finalizados primero.
b. Es fácil de implementar.
c. Es una planificación con starvation free.

Desventajas:

a. No es escalable, ya que si se debe recorrer siempre por cada tick la cola de baja prioridad sería muy costoso si existiesen muchos procesos.
b. No es muy eficiente por lo anteriormente explicado.

# 4 Batería de pruebas

A continuación, se presentan las pruebas realizadas para comprobar el correcto funcionamiento de los planificadores implementados y descritos anteriormente.

Ejemplo:

- **RR_01 – Un hilo**: Prueba 01 para planificador RR con un hilo.

Algunas salidas obtenidas de las pruebas ocupan significativamente, para realizar una síntesis de la salida obtenida se deberán tener en cuenta la siguiente codificación:

- **SWAPCONTEXT FROM X TO Y**: X→Y.
- **THREAD X FINISHED**: X *FINISHED.*
- **THREAD X TERMINATED: SETCONTEXT OF Y**: *TERMINATED* X→Y

| RR_01 – Un hilo | |
|---|---|
| **Objetivo** | Ver que solo ejecuta el hilo por defecto (0) correctamente. Probar el caso límite inferior. |
| **Entradas** | Prioridad para el hilo por defecto. |
| **Salida** | *** THREAD 0 FINISHED<br>FINISH<br>MYTHREAD_FREE: NO THREAD IN THE SYSTEM<br>EXITING... |
| **Conclusión** | El hilo por defecto ejecuta únicamente y después finaliza. |

**Tabla 1 – RR_01**

| RR_02 – Diez hilos | |
|---|---|
| **Objetivo** | Ver la correcta ejecución de los hilos en el orden de creado. Observando el orden en el que finalizan según el tiempo de ejecución de cada uno. |
| **Entradas** | Diez hilos. Los hilos 2, 5 y 9 son los que menos tiempo tardan en ejecutar, seguidos de 0, 3, 6 y 7, finalmente los que más tardan en ejecutar son 1, 4 y 8. Su lanzamiento es en orden secuencial, con un *delay* entre los hilos 3 y 4, y los hilos 8 y 9. |
| **Salida** | 0→1, 1→2, 2→3, 3→0 (al no lanzarse los demás hilos). Tras esto se lanzan los demás hilos hasta el 8.<br>0→1, 1→2. 2 *FINISHED* y *TERMINATED* 2 → 3.<br>3→4, 4→5, 5→6, 6→7, 7→8 y 8→0. Y después se lanza el hilo 9 con id=2, porque el hilo 2 ya ha finalizado.<br>0 *FINISHED*, *TERMINATED* 0→1, 3 *FINISHED*, *TERMINATED 3→4, 4→5, 5 FINISHED, 5 TERMINATED 5→6, 6→7, 7→8, 8→2, 2→1, 1 FINISHED, TERMINATED* 1→4, 4→6, 6 *FINISHED, TERMINATED* 6→7, *7 FINISHED, TERMINATED 7→8,* 8→2, 2 *FINISHED, TERMINATED 2→4, 4 FINISHED, TERMINATED 4→8* y finalmente 8 *FINISHED.* |
| **Conclusión** | En esta prueba se puede observar que se realizan *SWAP* de manera correcta en el orden de lanzamiento de los hilos. Sabiendo el tiempo de ejecución de cada hilo (como se ha mencionado anteriormente), sabemos el orden de finalización: 2, 0, 3, 5, 1, 6, 7, 2 (siendo este el 9), 4 y 8. Al finalizar un hilo antes de que otro se lance, se puede observar que se realiza correctamente la asignación de *tid* y que se encola según haya llegado. |

**Tabla 2 – RR_02**

| RRF_01 – Diez hilos | |
|---|---|
| **Objetivo** | 1. Ver la correcta ejecución de los hilos según sus prioridades. Y según el orden de lanzamiento para prioridades altas, y para prioridades bajas.<br>2. Ver que al llegar un proceso de baja prioridad no ejecuta si hay un proceso de alta prioridad que llega después de él. Aunque el proceso anterior sea de baja o alta prioridad.<br>3. Ver que los hilos de alta prioridad ejecutan hasta finalizar su tiempo de cómputo total, se asignan tiempos variables para observar el comportamiento.<br>4. Ver la inanición de los hilos de baja prioridad cuando el resto de hilos son de alta prioridad. |
| **Entradas** | Diez hilos. Los hilos 1 y 8 son de baja prioridad, y el tiempo de cómputo es grande. Los demás, incluido el de por defecto, son de alta prioridad. El tiempo de cómputo de estos últimos son variables. |
| **Salida** | Comienza ejecutando el hilo por defecto que tiene prioridad alta. A continuación, se lanza el hilo 1 con baja prioridad que no ejecuta. Ejecuta hasta acabar el hilo 0, 2, 3, 4, 5, 6, 7, 9 después el hilo 1 va haciendo *SWAP* con el hilo 8 hasta finalizar.<br>Lanzamiento hilo 0, hilo 1, 2, 3, 4, 5, 6, 7 y 9.<br>*FINISHED 0, TERMINATED 0→2, FINISHED 2, TERMINATED 2→3, FINISHED 3, TERMINATED 3→4, FINISHED 4, TERMINATED 4→5, FINISHED 5, TERMINATED 5→6, FINISHED 6, TERMINATED 6→7, FINISHED 7, TERMINATED 7→9, FINISHED 9, TERMINATED 9→1,* 1→8, 8→1, 1→8, 8→1, 1→8, 8→1*, FINISHED 1, TERMINATED 1→8,*<br>*FINISHED 8* |
| **Conclusión** | En esta prueba se puede observar que al llegar hilos de baja prioridad no ejecutan si hay hilos de alta prioridad ejecutando o en la cola. Además de que se produce inanición al haber muchos hilos de alta prioridad. Ejecutan los procesos de baja prioridad al final de todos los hilos de alta prioridad. |

**Tabla 3 – RRF_01**

| RRF_02 – Cuatro hilos | |
|---|---|
| **Objetivo** | 1. Ver que los hilos de prioridad baja son expulsados de la CPU al llegar un hilo de prioridad más alta que él.<br>2. Ver que los hilos de alta prioridad ejecutan antes de los de baja, aunque lleguen a ser lanzados más tarde. |
| **Entradas** | Cuatro hilos, el primero de baja prioridad que es el hilo por defecto. El segundo de alta prioridad, el tercero de baja prioridad y el último de alta prioridad. |
| **Salida** | Lanzamiento de hilo 0 de baja prioridad y que entra en ejecución.<br>Lanzamiento de hilo de alta prioridad y que expulsa al hilo 0.<br>**** THREAD 0 EJECTED : SET CONTEXT OF 1, 1 FINISHED, TERMINATED 1→0, *** THREAD 0 EJECTED : SET CONTEXT OF 2, 2 FINISHED, TERMINATED 2→1,1→0, 0 FINISHED,*<br>*TERMINATED 0→1, 1→2, 2 FINISHED, TERMINATED 2→1,*<br>*1 FINISHED* |
| **Conclusión** | Los hilos de alta prioridad al ser lanzados, expulsan a los de baja prioridad, pero no a los de alta prioridad. Además, todos los hilos de baja prioridad al ser lanzados, no ejecutan si hay un hilo de alta prioridad en la cola FIFO de alta prioridad. |

**Tabla 4 – RRF_02**

| RRFI_01 – Cuatro hilos | |
|---|---|
| **Objetivo** | 1. Ver que los hilos de alta prioridad ejecutan antes de los de baja, aunque lleguen a ser lanzados más tarde.<br>2. Ver que los hilos de baja prioridad tras un tiempo de inanición pasan a la cola de alta prioridad.<br>3. Ver que los hilos vuelven a la cola de baja prioridad tras ejecutar.<br>4. Ver que si un hilo de baja prioridad que ha sido subido a alta prioridad, y después de él otro, el segundo no expulsa al primero de la CPU. |
| **Entradas** | Siete hilos, de los cuales los hilos 0, 2, 3 y 4 son de alta prioridad, el resto de baja prioridad. El lanzamiento es secuencial. |
| **Salida** | Lanzamiento de hilo 0 de alta prioridad y que entra en ejecución. Lanzamiento de hilo 1, 2, 3, 4, 5, 6 y 7.<br>*FINISHED 2, TERMINATED 2→3,*<br>*1 MOVED TO HIGH PRIORITI QUEUE, FINISHED 3,*<br>*TERMINATED 3→4,*<br>*5 MOVED TO HIGH PRIORITY QUEUE, 6 MOVED TO HIGH PRIORITY QUEUE, 7 MOVED TO HIGH PRIORITY QUEUE, FINISHED 4,*<br>*TERMINATED 4→1,1→5, 5→6, 6→7, 7→1, 1→5, 5→6, 6→7, 7→1, 1→5, 5→6, 6→7, 7→1, FINISHED 1, TERMINATED 1→5,*<br>*FINISHED 5, TERMINATED 5→6, FINISHED 6, TERMINATED 6→7, FINISHED 7.* |
| **Conclusión** | En esta prueba se puede observar como los procesos de baja prioridad al sufrir inanición son movidos a la cola de alta prioridad para poder ejecutar, sin embargo, cuando dos procesos son de baja prioridad y son movidos a la cola de alta prioridad, el segundo no expulsa al primero. |

**Tabla 5 – RRFI_01**

# 5 Conclusiones

Como hemos ido explicando en el presente documento, nos encontrábamos con unas políticas de planificación sencillas de implementar, pero ninguna sería viable para un sistema actual, ya que en la actualidad hay cientos de procesos de distinta prioridad y muchos sufrirían starvation o por el caso contrario no serían consideradas sus prioridades.

Si queremos evitar la hambruna el método implementado no es eficiente, ya que recorrer una cola de cientos de procesos haría que se perdiera más tiempo en ello que en la propia ejecución del proceso, incrementando la variable ticks de manera no real, ya que el primer proceso recorrido debería tener más ticks al finalizar el recorrido de la cola.

Es por ello que es una primera aproximación a los distintos planificadores y como tratar con cada proceso en ellos.