Practica 1: Planificación de procesos

EPS

Ingeniería Informática

Diseño de Sistemas Operativos

Santiago Ramos Sevillano NIA 100383401 Gr. 83
100383401@alumnos.uc3m.es
Iván Valbuena Gálvez NIA 100383375 Gr. 83
100383375@alumnos.uc3m.es
Lucía Ruz Sáez NIA 100363940 Gr. 83
100363940@alumnos.uc3m.es

Santiago Ramos Sevillano NIA 100383401
Iván Valbuena Gálvez NIA 100383375
Lucía Ruz Sáez NIA 100363940

uc3m | Universidad Carlos III de Madrid

Índice

Índice de figuras

Índice de tablas

Santiago Ramos Sevillano NIA 100383401
Iván Valbuena Gálvez NIA 100383375
Lucía Ruz Sáez NIA 100363940

1 Introducción

A través del presente documento se van a definir los pasos seguidos para el desarrollo de **tres algoritmos de planificación de hilos en el espacio de usuario**, los cuales se encuentran escritos en el lenguaje de programación C. Estos planificadores han sido creados siguiendo, tanto las pautas dadas en el enunciado de la práctica, como aquellos conocimientos básicos impartidos en la asignatura *Diseño de Sistemas Operativos*.

En primer lugar, es necesario recordar que todos los **cambios de contexto** en la planificación de procesos de esta práctica se van a realizar en **espacio de usuario**.

El primer planificador es **Round-Robin**. El objetivo de esta política es que cada hilo ejecuta un número determinado de ticks definidos en la rodaja, y una vez terminado dicho número de paso al siguiente hilo listo para ejecutar. El documento final donde se encuentra el código de dicho planificador será nombrado como ***RR.c***.

Por otro lado, la segunda política ha de ser **Round-Robin**, para aquellos hilos con prioridad baja, y **Shortest Job First con prioridades**, para aquellos hilos de prioridad alta. La política Short Job First ejecuta, en primer lugar, aquellos trabajos más cortos frente a aquellos más largos. El documento donde se encuentra dicho fragmento del código se denomina ***RRS.c***.

Por último, el último planificador ha de utilizar nuevamente la política **Round-Robin** para aquellos hilos con prioridad baja, mientras en los hilos de prioridad alta han de utilizar **Shortest Job First con prioridades**. La diferencia con la segunda implementación se encuentra en que, para este caso, se ha de añadir como funcionalidad un posible **cambio de contexto voluntario**. En este caso, el documento donde se encuentra dicho fragmento de código se denomina ***RRSD.c***.

2 Diseño usado en el código

Para el diseño del código se ha analizado el problema inicial, y a continuación se indican las estructuras de datos y funciones implementadas para cada caso en pseudocódigo.

Es importante destacar que para la realización de nuestros planificadores nos hemos basado en el uso de dos variables globales, **running** (que ya venía definida en el código base) y **prev**, que es del mismo tipo que running. La idea es tener siempre un control tanto del hilo que sale de ejecutar (ya sea por fin de rodaja, fin de ejecución, expulsión o bloqueo del mismo) como del hilo que entra a ejecutar. Por lo tanto, **running** siempre almacenará el hilo al que le toca ejecutar (a excepción de cuando ejecute el hilo **idle**) y **prev** el que sale de ejecución, es decir, al realizar un cambio de contexto el hilo que estaba ejecutando **(running)** se almacenará en **prev** y **running** llamará al planificador para obtener el siguiente hilo a ejecutar. Debido a esta decisión de diseño no hemos hecho uso de la función ***mythread_gettid*** puesto que ya tenemos un control de los id de los hilos.

Además, es preciso mencionar que en la función ***mythread_setpriority***, al igual que se da unos *remaining_ticks* a los procesos de prioridad alta se lo hemos dado a los de prioridad baja, evitando casos en los que si el hilo 0 era de baja prioridad y era expulsado por uno de alta prioridad o llamaba a *read_disk*, su tiempo se acababa y era expulsado. (No obstante haremos pruebas para comprobar que la expulsión de hilos funciona correctamente)

Nota: antes de acceder a cualquiera de las colas generadas, ya sea para encolar o desencolar, se procederá a desactivar las interrupciones de reloj y disco en ese orden. Tras realizar la operación con la cola se reactivará la

interrupción de disco y luego la de reloj, en ese orden. Esta medida la hemos realizado para los 3 casos que se proponen, aunque en los dos primeros no se use la interrupción de disco.

Las funciones *mythread_exit()*, *mythread_timeout()* y *activator(TCB *next)* son iguales en los códigos de los 3 planificadores, con lo cual, para ahorrar espacio y evitar repetir innecesariamente, hemos decidido ponerlas al principio:

- **mythread_exit():**
  - Estado del proceso que ha terminado = FREE
  - Liberar espacio asignado al proceso que acaba
  - Llamada al planificador (scheduler)
  - Activator (hilo a ejecutar)
- **mythread_timeout(int tid):**
  - Estado del proceso que ha terminado = FREE
  - Liberar espacio asignado al proceso expulsado
  - Llamada al planificador (scheduler)
  - Activator (hilo a ejecutar)
- **activator (TCB *next):**
  - Si el proceso anterior ha acabado o ha sido expulsado por timeout (su estado es FREE):
    - setcontext (nextContext)
  - Para el resto de los casos:
    - swapcontext (prevContext, nextContext)

2.1 Round Robin

Una vez analizado el enunciado inicial, para poder implementar el planificador Round-Robin en necesario añadir el campo que contabilice los ticks, es decir, las rodajas de tiempo en el BCP, y también es necesario modificar la interrupción de reloj. Por ello:

2.1.1 Estructuras de datos

- Ticks en el BCP para poder tener en cuenta las rodajas de tiempo.
- Cola de procesos. En este caso se reutilizará la cola ofrecida en el enunciado de la práctica *"t_queue"*.

2.1.2 Funciones

- **mythead_create (void (*fun_addr)( ), int priority, int seconds)**
  - Crea un nuevo proceso
  - El número de ticks de su BCP son el máximo por rodaja (BCP. ticks = QUANTUM_TICKS)
  - El estado pasa a estar listo (BCP.state = INIT)
  - Insertar en la cola de procesos
- **TCB *scheduler ( )**
  - Si no está vacía la lista de listos:
    - Desencola el primer proceso y lo guarda
    - Devuelve el proceso
- **timer_interrupt ( )**
  - Running.ticks -= 1
  - running.remaining_ticks -= 1
  - Si running.remaining_ticks < 0:

- Entonces el hilo debe acabar por timeout y se llama a dicha función pasando el tid del hilo a expulsar.

- Si running.ticks <= 0:
  - El estado del hilo que estaba ejecutando pasa a listo para ejecutar.
  - running.ticks = QUANTUM_TICKS
  - Encolar en cola de procesos
  - prev = hilo ejecutado hasta ahora (running)
  - Llama al planificador (scheduler)
  - Si el hilo al que le toca ejecutar es distinto al anterior que ejecutó:
    - activator (hilo a ejecutar)

2.2 Round-Robin/ SJF con prioridades

Para poder implementar el planificador *Round-Robin/SJF* en necesario añadir el campo que contabilice los ticks, es decir, las rodajas de tiempo en el BCP, y también es necesario modificar la interrupción de reloj. Por ello:

2.2.1 Estructuras de datos

- Ticks en el BCP: para poder tener en cuenta las rodajas de tiempo.
- Cola de procesos: llamada “t_queue” para albergar los procesos de baja prioridad.
- Cola de procesos de alta prioridad: llamada “t_queue_high” para albergar los procesos de alta prioridad ordenados de menor a mayor según su duración (remaining_ticks).

2.2.2 Funciones

- **mythead_create (void (*fun_addr)( ), int priority, int seconds):**
  - Crea un nuevo proceso
  - El número de ticks de su BCP son el máximo por rodaja (BCP. ticks = QUANTUM_TICKS)
  - El estado pasa a ser listo (BCP.state = INIT)
  - Si el nuevo proceso es de prioridad baja:
    - Insertar en cola de baja prioridad “t_queue”.
  - Si el nuevo proceso es de prioridad alta y el proceso ejecutándose (running) no:
    - Estado de running = INIT
    - Ticks de running = QUANTUM_TICKS
    - Running se encola en la cola de baja prioridad “t_queue”
    - prev = running
    - running = nuevo proceso
    - activator (hilo a ejecutar)
  - Si el nuevo proceso es de prioridad alta, running también lo es y el proceso es más corto que el tiempo restante de running:
    - Estado de running = INIT
    - Se encola running en la cola de alta prioridad “t_queue_high”
    - prev = running
    - running = nuevo proceso
    - activator (hilo a ejecutar)
  - Si el nuevo proceso es de prioridad alta, running también y el proceso es más largo que el tiempo restante de running:
    - Estado del nuevo proceso = INIT
    - Se encola el nuevo proceso en la cola de alta prioridad “t_queue_high”

- **TCB *scheduler ( )**:
  - Si no está vacía la cola de listos de alta prioridad “t_queue_high”:
    - Desencola el primer proceso de esta cola

        - Devuelve el proceso desencolado
    - Else, si no está vacía la cola de listos de baja prioridad “t_queue”:
        - Desencola el primer proceso de esta cola
        - Devuelve el proceso

- **timer_interrupt ( )**:
    - Si running es de baja prioridad:
        - ticks de running -= 1
    - remaining_ticks de running -= 1
    - Si remaining_ticks de running < 0:
        - Entonces el hilo debe acabar por timeout y se llama a dicha función pasando el tid del hilo a expulsar
    - Si ticks de running = 0 y running es de baja prioridad:
        - El estado de running pasa a listo para ejecutar
        - Ticks de running = QUANTUM_TICKS
        - Encolar en cola de baja prioridad “t_queue”
        - prev = hilo ejecutado hasta ahora (running)
        - Llama al planificador (scheduler)
        - Si el hilo al que le toca ejecutar es distinto al anterior que ejecutó:
            - activator (hilo a ejecutar)

2.3 Round-Robin/ SJF con posibles cambios de contexto voluntarios

2.3.1 Estructuras de datos

- Ticks en el BCP: para poder tener en cuenta las rodajas de tiempo.
- Cola de procesos: llamada “t_queue” para albergar los procesos de baja prioridad.
- Cola de procesos de alta prioridad: llamada “t_queue_high” para albergar los procesos de alta prioridad ordenados de menor a mayor según su duración (remaining_ticks).
- Cola de bloqueados: llamada “t_queue_wait” para albergar los procesos bloqueados.

2.3.2 Funciones

- **mythead_create (void (*fun_addr)( ), int priority, int seconds):**
    - Crea un nuevo proceso
    - El número de ticks de su BCP son el máximo por rodaja (BCP. ticks = QUANTUM_TICKS)
    - El estado pasa a ser listo (BCP.state = INIT)
    - Si el nuevo proceso es de prioridad baja:
        - Insertar en cola de baja prioridad “t_queue”.
    - Si el nuevo proceso es de prioridad alta y el proceso ejecutándose (running) no:
        - Estado de running = INIT
        - ticks de running = QUANTUM_TICKS
        - running se encola en la cola de baja prioridad “t_queue”
        - prev = running
        - running = nuevo proceso
        - activator (hilo a ejecutar)
    - Si el nuevo proceso es de prioridad alta, running también lo es y el proceso es más corto que el tiempo restante de running:
        - Estado de running = INIT
        - Se encola running en la cola de alta prioridad “t_queue_high”
        - prev = running
        - running = nuevo proceso
        - activator (hilo a ejecutar)
    - Si el nuevo proceso es de prioridad alta, running también y el proceso es más largo que el tiempo restante de running:

        - Estado del nuevo proceso = INIT
        - Se encola el nuevo proceso en la cola de alta prioridad “t_queue_high”

- **read_disk():**
    - Si data_in_page_cache() distinto de 0:
        - estado de running = WAITING
        - Si running es de prioridad baja:
            - ticks de running = QUANTUM_TICKS
        - Se encola running en la cola de bloqueados “t_queue_wait”.
        - prev = running
        - LLamamos al scheduler para conseguir el próximo proceso a ejecutar.
        - activator (hilo a ejecutar)

- **disk_interrupt(int sig)**:
    - Si la cola de bloqueados no está vacía:
    - Se desencola el primer proceso de la cola de bloqueados “t_queue_wait”
    - Si el proceso es de prioridad baja:
        - Se encola el proceso en la cola de prioridad baja “t_queue”
    - Else:
        - Se encola el proceso en la cola de prioridad alta “t_queue_high”

- **TCB *scheduler ( ):**
    - Si las colas de procesos de alta y baja prioridad están vacías y la cola de bloqueados no:
        - Devolver el proceso idle
    - Si no está vacía la cola de listos de alta prioridad “t_queue_high”:
        - Desencola el primer proceso de esta cola
        - Devuelve el proceso desencolado
    - Else, si no está vacía la cola de listos de baja prioridad “t_queue”:
        - Desencola el primer proceso de esta cola
        - Devuelve el proceso

- **timer_interrupt ( ):**
    - Si running es de baja prioridad:
        - ticks de running -= 1
    - remaining_ticks de running -= 1
    - Si remaining_ticks de running < 0 y prioridad de running es distinta de SYSTEM (para no expulsar al idle):
        - Entonces el hilo debe acabar por timeout y se llama a dicha función pasando el tid del hilo a expulsar
    - Si id de running igual a -1 (es el idle) y alguna de las colas de procesos no está vacía:
        - prev = running
        - Llama al planificador (scheduler)
        - activator (hilo a ejecutar)
    - Si la cola de alta prioridad no está vacía y la prioridad de running es alta:
        - Si el tiempo del primer proceso de esa cola es menor que el tiempo restante de running:
            - estado de running = INIT
            - running se encola en la cola de alta prioridad “t_queue_high”
            - prev = running
            - Llama al planificador (scheduler)
            - activator (hilo a ejecutar)
    - Si la cola de alta prioridad no está vacía y la prioridad de running es baja:
        - estado de running = INIT
        - ticks de running = QUANTUM_TICKS
        - running se encola en la cola de baja prioridad “t_queue”

- prev = running
- Llama al planificador (scheduler)
- activator (hilo a ejecutar)

- Si ticks de running = 0 y running es de baja prioridad:
  - El estado de running pasa a listo para ejecutar
  - ticks de running = QUANTUM_TICKS
  - Encolar en cola de baja prioridad “t_queue”
  - prev = hilo ejecutado hasta ahora (running)
  - Llama al planificador (scheduler)
  - Si el hilo al que le toca ejecutar es distinto al anterior que ejecutó:
    - activator (hilo a ejecutar)

3 Batería de pruebas

Para poder comprobar el código realizado se van a crear una serie de **diagramas para cada planificador**, para poder ir viendo las distintas opciones de ejecución. Cada diagrama contendrá todos los posibles casos como nodos terminales. Se han incluido líneas discontinuas para aquellos casos que se puedan repetir, y así poder cerrar el ciclo total de cada ejecución.

En la clase principal del programa, *main.c*, se han diferenciado todas las posibles pruebas de tal manera que para ejecutar cada una individualmente se hará de la siguiente manera: ***./main testX***. Para una mayor comodidad, todas las pruebas se llaman testX, donde X es el número identificativo de cada prueba.

A lo largo de este punto se irá desarrollando que hace cada prueba, indicando a su vez el nombre otorgado.El ***test0*** es el test por defecto que se da en el código base de la práctica.

3.1 Round Robin

Este caso es el más sencillo a la hora de crear el diagrama, ya que no se tiene en cuenta la prioridad del proceso, sino que se encolan todos de igual manera y se van ejecutando en función a una rodaja de tiempo predeterminada. Por ello, el diagrama sera el indicado en la Figura 1.

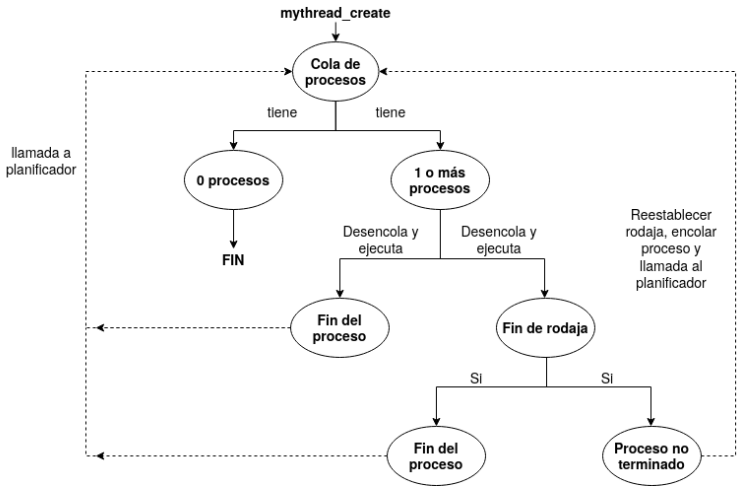


Figura 1: Árbol de pruebas Round Robin.

Como se puede ver, hay 2 opciones: que la **cola de procesos está vacía** o tenga algún proceso. En caso de estar vacía, **termina la ejecución**. Por otro lado, en caso de tener 1 o más procesos se ejecuta cada proceso el tiempo indicado en QUANTUM_TICKS. En caso de terminar el proceso antes de finalizar dicha rodaja de tiempo, finaliza la ejecución y llamará al planificador para obtener el siguiente proceso si quedan (se implementa como ***test1***. Tabla 2).

En caso de **finalizar** el tiempo marcado por cada **rodaja** pueden darse dos casos: que se haya terminado de ejecutar el hilo, a la par que termina el tiempo de la rodaja (caso implementado como ***test2***. Tabla 2), o que aún no haya terminado su ejecución (caso ya implementado en el ***test1***. Tabla 2). En el primer caso, se terminará la ejecución de dicho hilo y se procederá a **llamar al planificador para obtener el siguiente proceso si quedan**. En el segundo caso, el proceso volverá a poner el número de rodaja en su totalidad y será **encolado** en la cola de procesos, procediendo a llamar, nuevamente, al planificador.

Si llega un proceso durante la ejecución de otro proceso se encola con su rodaja correspondiente en la cola de procesos (comprobado en todos los test anteriores, principalmente en el ***test1***. Tabla 2).

3.2 Round-Robin/ SJF con prioridades

El diagrama creado para este caso ha sido el mostrado en la Figura 2.

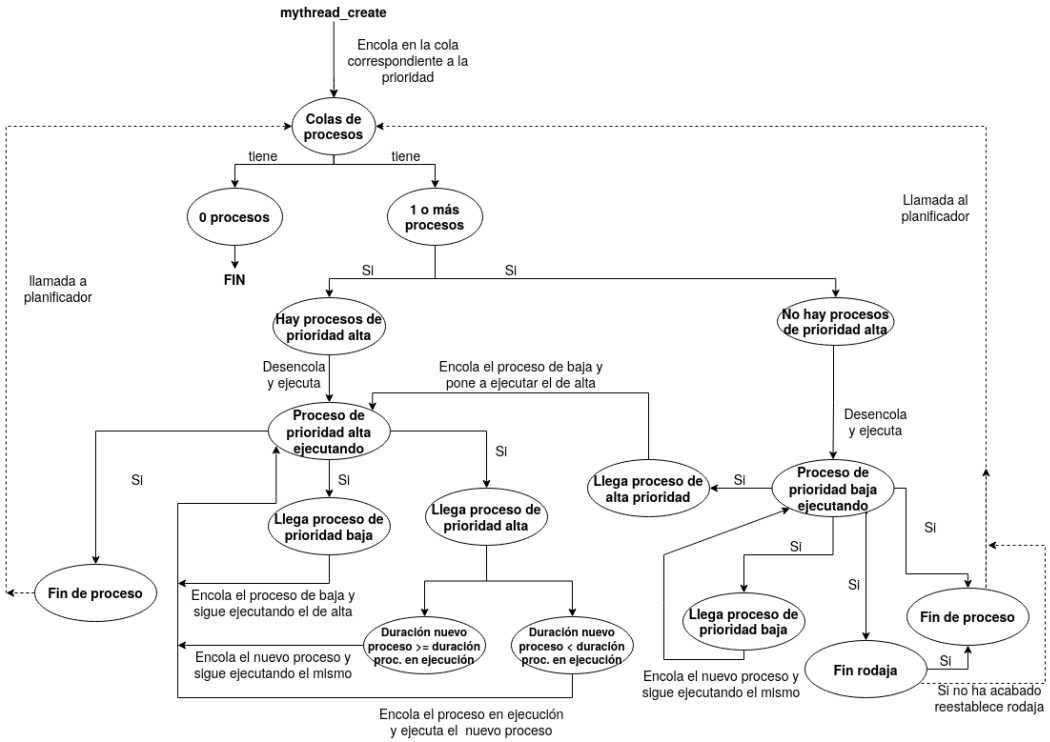


Figura 2: Árbol de pruebas RRS

En primer lugar, se analizan la cola de alta prioridad y la de baja prioridad, en ese orden, para ver qué proceso va a ejecutarse (llamada al planificador). En caso de no haber procesos pendientes finaliza la ejecución. En el caso de existir algún proceso pendiente, en **primer lugar** se ejecutarán aquellos procesos de **alta prioridad**.

Si mientras se ejecuta un proceso de alta prioridad entra un proceso de **baja prioridad**, se mete este último en la cola de baja prioridad y **continúa** ejecutándose el primer proceso (caso implementado como ***test3***. Tabla 3). En caso de entrar otro proceso de **alta prioridad**, sería necesario **comparar la duración** de este. Si la duración del nuevo proceso es **mayor** o igual que la duración restante del proceso en ejecución, se encola el nuevo proceso en la cola de alta prioridad y se **continúa** con la ejecución del proceso inicial (caso también comprobado en ***test3***. Tabla 3). Si por el contrario la duración es menor, se realiza un cambio de contexto, por lo que el proceso que acaba de llegar pasa a ejecución y el que estaba ejecutando se encola en la cola de alta prioridad (caso implementado como ***test4***. Tabla 3). Si termina la ejecución del proceso se vuelve a llamar al planificador para obtener el siguiente proceso si quedan.

Por otro lado, si al analizar ambas colas no hay procesos de alta prioridad pero sí de **baja prioridad**, pasan a ejecutarse en orden según la política **Round Robin**. En caso de encontrarse ejecutando un proceso de **baja prioridad** y entrar otro de la misma prioridad, este se **encolaría** en la cola de baja prioridad para ser ejecutado en su momento (caso implementado como ***test5***. Tabla 3). Si el proceso que se está ejecutando consume el tiempo total de la rodaja, se encola reiniciando nuevamente dicho valor y se vuelven a analizar las colas. También se analiza las colas cuando el proceso en ejecución haya terminado.

Además, mientras se ejecuta un proceso de baja prioridad puede darse el caso en el que entre un proceso de **alta prioridad**. En este caso, dejará de ejecutarse el proceso actual, reiniciando la rodaja y encolándose en la cola de baja prioridad. En ese momento se **comenzaría a ejecutar** el proceso de alta prioridad (caso también implementado como ***test5***. Tabla 3).

3.3 Round-Robin/ SJF con posibles cambios de contexto voluntarios

Para el siguiente diagrama se ha partido del diagrama adjunto en el apartado anterior. Figura 3

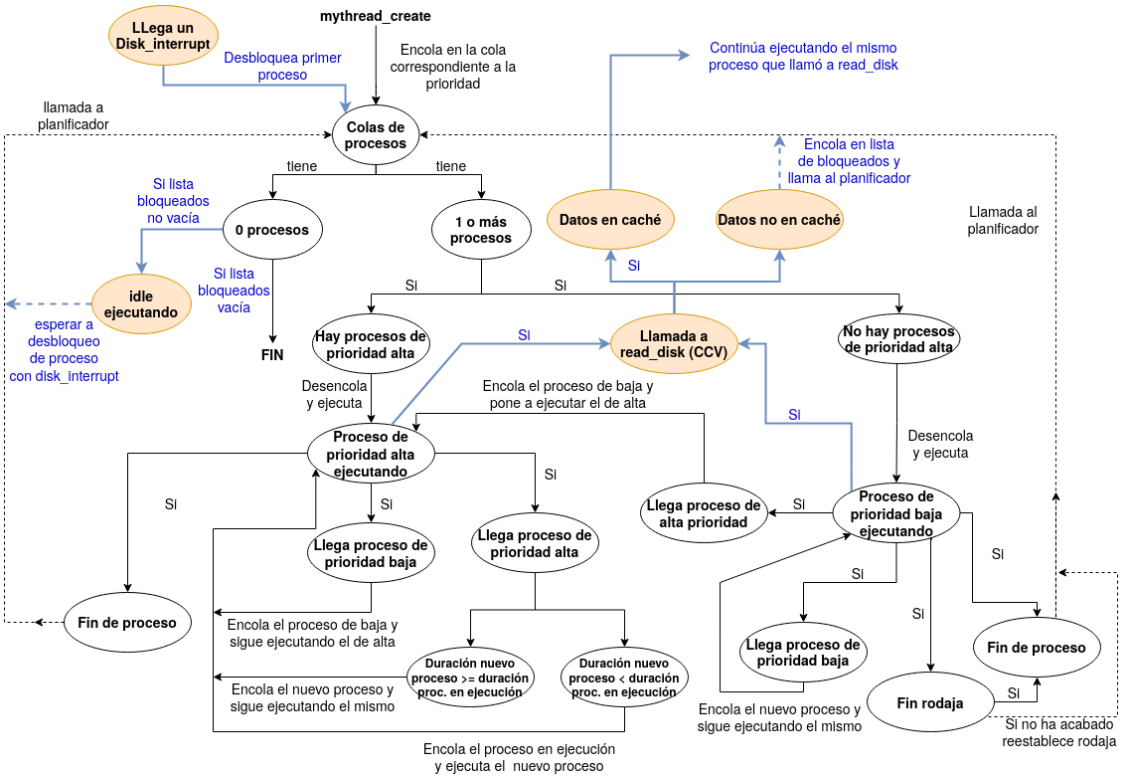


Figura 3: Árbol de pruebas RRSD

Las diferencias se encuentran marcadas de otro color: naranja para los nodos y azul para los diversos enlaces y sus respectivos textos asociados. Como se puede apreciar, se han añadido los casos para los **cambios de contexto voluntarios** (indicados en la figura como CCV). Puesto que el código también se reutiliza, aquellas pruebas que resulten repetitivas no se han ejecutado.

Durante la ejecución de un proceso este puede requerir un cambio de contexto voluntario (llamada a *read_disk*). Si la información que requiere el proceso se encuentra **en caché**, el proceso tomará dichos datos y **continuará** su ejecución sin realizar ninguna acción más (caso implementado como ***test6*** este caso vamos a probar que se realiza correctamente tanto para hilos de alta como baja prioridad. Tabla 4), dependiendo si parte de un proceso de prioridad alta o de prioridad baja). Si por el contrario la información requerida **no** se encontrase **en memoria caché**, el proceso se añadiría a la **cola de bloqueados** y se llamaría al planificador para ejecutar el siguiente proceso que toque (caso implementado como ***test7***. Tabla 4).

En caso de no haber más procesos a ejecutar en las colas de alta y baja prioridad, si la cola de bloqueados no se encuentra vacía se pondrá en ejecución el **thread *idle***. Este thread ejecutará un bucle infinito, de tal forma que el algoritmo consulte cada tick de reloj si existe algún thread listo para ejecutar, tal y como se indica en el enunciado de la práctica (caso implementado como ***test8***. Tabla 4). Además, observamos que existe la interrupción *disk_interrupt* que será la encargada de desbloquear los procesos de la lista de bloqueados, sacando al primero de ellos de la cola cuando se llama a la función.

4 Resultados batería de pruebas

A continuación se van a presentar, en 3 tablas diferentes, las pruebas realizadas para cada uno de los apartados pedidos, presentando información como la prueba a realizar, el objetivo de la misma y el resultado obtenido. Además, habrá una cuarta tabla con pruebas comunes, cuyo resultado es el mismo en los 3 planificadores (vamos a ejecutarlas en el planificador RRSD, puesto que es el más complejo y agrupa todo lo de los dos anteriores).

Prueba	Objtivo de la prueba	Resultado

Test9: vamos a crear un hilo de alta prioridad y otro de baja prioridad, siendo el hilo 0 de alta prioridad	El objetivo de esta prueba es comprobar que tanto los hilos de alta como baja prioridad son expulsados si exceden el tiempo de ejecución asignados. Para ello vamos a incluir en su función un bucle for que produzca un tiempo de ejecución mayor al propuesto, por lo que deben ser expulsados. Para el caso del RR no importa la prioridad.	`*** THREAD 0 FINISHED`
		`*** THREAD 0 TERMINATED : SETCONTEXT OF 1`
		`*** THREAD 1 EJECTED`
		`*** THREAD 2 EJECTED`
		`*** FINISH`

Test10: el hilo 0 no va a crear ningún hilo	El objetivo es comprobar el caso en el que solo existe el hilo 0, el cual debe acabar y terminarse la ejecución	`*** THREAD 0 FINISHED` `*** FINISH`.

Tabla 1: Pruebas comunes

Prueba	Objtivo de la prueba	Resultado

Test1: vamos a crear hilos cuyo tiempo de ejecución sea menor a la rodaja y otros con un tiempo de ejecución mayor a la rodaja	Observar que los hilos cuyo tiempo de ejecución es inferior a la rodaja, cuando reciben el turno de ejecución, acaban su ejecución entera y realizan un exit sin llegar a hacer un swapcontext cediendo el turno (pues su rodaja aún no ha acabado), mientras que los otros hilos se irán cediendo el turno de ejecución hasta acabar.	`*** THREAD 0 FINISHED`
		`*** THREAD 0 TERMINATED : SETCONTEXT OF 1`
		`*** THREAD 1 FINISHED`
		`*** THREAD 1 TERMINATED : SETCONTEXT OF 2`
		`*** SWAPCONTEXT FROM 2 TO 3`
		`*** THREAD 3 FINISHED`
		`*** THREAD 3 TERMINATED : SETCONTEXT OF 4`
		`*** SWAPCONTEXT FROM 4 TO 5`
		`*** SWAPCONTEXT FROM 5 TO 2`
		`*** SWAPCONTEXT FROM 2 TO 4`
		`*** SWAPCONTEXT FROM 4 TO 5`
		`*** SWAPCONTEXT FROM 5 TO 2`
		`*** SWAPCONTEXT FROM 2 TO 4`
		`*** SWAPCONTEXT FROM 4 TO 5`
		`*** SWAPCONTEXT FROM 5 TO 2`
		`*** SWAPCONTEXT FROM 2 TO 4`
		`*** SWAPCONTEXT FROM 4 TO 5`
		`*** SWAPCONTEXT FROM 5 TO 2`
		`*** SWAPCONTEXT FROM 2 TO 4`
		`*** SWAPCONTEXT FROM 4 TO 5`
		`*** SWAPCONTEXT FROM 5 TO 2`
		`*** THREAD 2 FINISHED`
		`*** THREAD 2 TERMINATED : SETCONTEXT OF 4`
		`*** THREAD 4 FINISHED`
		`*** THREAD 4 TERMINATED : SETCONTEXT OF 5`
		`*** THREAD 5 FINISHED`
		`*** FINISH`

Test2: vamos a calcular los ticks que debemos poner para que estos coincidan con el valor de la rodaja, creando un hilo con esos valores de ticks	Comprobar que ,si un hilo acaba su ejecución justo cuando la rodaja tiene valor 0, éste sale de la ejecución con un exit y no restablece su valor de rodaja y permanece en el sistema, lo cual sería un error.	`** THREAD 0 FINISHED`
		`** THREAD 0 TERMINATED : SETCONTEXT OF 1`
		`** THREAD 1 FINISHED`
		`** FINISH`

Tabla 2: Pruebas para RR

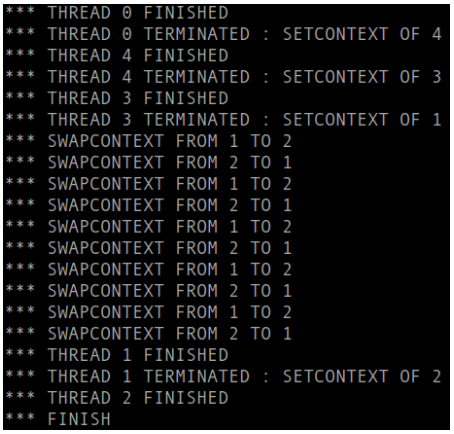
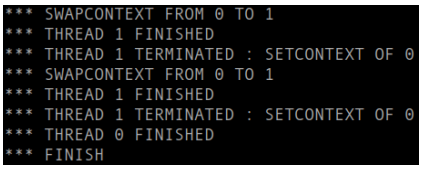
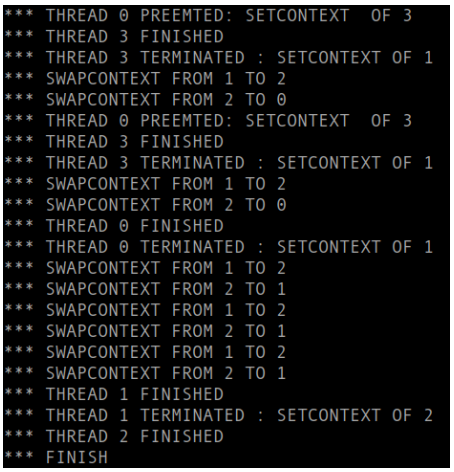
Prueba	Objtivo de la prueba	Resultado
Test3: vamos a poner la prioridad del hilo 0 como alta. Tras esto vamos a generar algunos hilos de baja prioridad y otros de alta prioridad y cuyo tiempo de ejecución será mayor que el del main	Queremos comprobar que los hilos de baja prioridad, aunque se van a crear antes que los de alta, se ejecutarán después de estos (y respetando las rodajas). Además, crearemos 2 hilos de alta prioridad de tal manera que el último creado tendrá un menor tiempo de ejecución que el otro, comprobando que éste acaba antes pero siempre después del main, pues le hemos puesto un tiempo mayor que el del hilo 0. Esto último nos permite comprobar el caso en el que hay un hilo de prioridad alta y llega otro de prioridad alta pero con un tiempo de ejecución mayor que lo que le queda al que está ejecutando, en cuyo caso no debe cambiarse el contexto	
Test4: vamos a poner la prioridad del hilo 0 como alta. Tras esto vamos a crear algunos hilos de prioridad alta con un tiempo de ejecución muy pequeño	El objetivo es que los hilos de alta prioridad generados tengan un tiempo de ejecución tan pequeño que sustituyan al hilo 0 en la ejecución. Así crearemos 2 hilos esperando que ambos acaben de ejecutar antes que el hilo 0. En este caso vamos a ponerles un tiempo de ejecución de 0.1 a los hilos de alta prioridad creados	
Test5: vamos a poner la prioridad del hilo 0 como baja. Tras esto se generarán hilos de baja prioridad y luego de alta prioridad (en ese orden)	El objetivo es observar que, si hay un hilo de prioridad baja ejecutando y llega uno de prioridad baja, los hilos se ejecutan según un RR, respetando las rodajas. Sin embargo, cuando llega uno de prioridad alta, el hilo de prioridad baja deja de ejecutar en favor del otro. Para esto generaremos varios hilos de prioridad baja en primer lugar y luego varios de prioridad alta. Debemos observar que el hilo 0 creará todos los hilos hasta llegar al primero de alta prioridad, el cuál le quitará la ejecución. Luego se ejecutarán los hilos de baja prioridad según un RR hasta crear el siguiente de prioridad alta que les quite de ejecución y así hasta acabar.	

Tabla 3: Pruebas para RRS

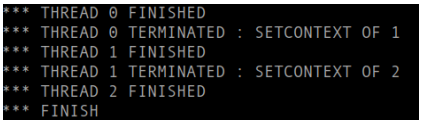
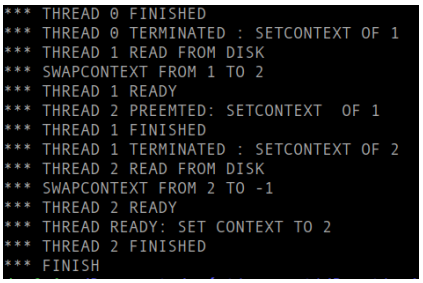
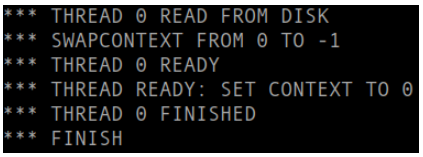
Prueba	Objtivo de la prueba	Resultado
Test6: vamos a crear el hilo 0 de alta prioridad. Tras esto se van a crear un hilo de alta prioridad y otro de baja prioridad que van a llamar a read_disk	Queremos comprobar cómo tanto los hilos de baja como alta prioridad, si llaman a read_disk y obtienen que el dato está en caché, siguen con su ejecución con normalidad. Para ello vamos a incluir una llamada a read_disk en la función de los hilos, para comprobar que ninguno de ellos se bloquea cuando los datos están en caché. El problema de esta prueba es que dependemos del dato recibido en la función data_in_page_cache (si se quiere probar solo una de las prioridades es tan sencillo como comentar la creación del hilo que no se quiera)	
Test7: vamos a crear el hilo 0 de alta prioridad. Tras esto se van a crear un hilo de alta prioridad y otro de baja prioridad que van a llamar a read_disk	Esta prueba es exactamente igual que la anterior pero queremos observar un comportamiento distinto. En este caso queremos comprobar que los hilos de ambas prioridades, si llaman a read_disk y los datos no están en caché, quedarán bloqueados hasta la siguiente interrupción de disco. El problema de la prueba es el mismo, se tiene que dar que ninguno de los datos esté en caché y depende de data_in_page_cache	
Test8: vamos a crear el hilo 0, con el cuál llamaremos aread_disk	El objetivo es llamar a read_disk con el hilo 0 y que esté se bloquee (porque los datos no están en caché). De esta forma tendremos todas las colas de procesos vacías menos la de bloqueados, por lo que el hilo idle se ejecutará hasta que el 0 se desbloquee (en pantalla se debe mostrar el cambio de contexto al hilo -1 y la impresión asociada a sacar al mismo de ejecución)	

Tabla 4: Pruebas para RRSD

Podemos comprobar que todas las pruebas son lo más completas posibles, evitando hacer pruebas independientes para casos muy simples que se puedan comprobar de manera conjunta en un mismo test. Así, se puede apreciar que hemos combinado ejecuciones que incluyen hilos de alta y baja prioridad cuando esto sea posible, explicando como debe ser el resultado obtenido. Además, al estar los planificadores más complejos basados en los anteriores, existen pruebas que no hemos repetido porque la ejecución es similar, como que la función mythread_exit funcione, y nos hemos enfocado en probar las funcionalidades individuales diferenciadoras de cada planificador.

5 Conclusiones

Gracias a esta práctica hemos aprendido cómo desarrollar distintos planificadores de procesos (Round-Robin, Round-Robin/SJF con prioridades y Round-Robin/SJF con cambios de contexto voluntarios), para lo que es necesario tener en cuenta, entre otras cosas, todas las estructuras de datos que necesita cada planificador (rodaja, colas de diferentes prioridades, etc.). También es necesario estudiar todos los casos posibles con detenimiento ya que, por ejemplo, no expulsar a un proceso de ejecución en su momento puede provocar errores en la ejecución que se pueden ir acumulando.

Por otra parte, esta práctica nos ha servido para mejorar nuestra capacidades de toma de decisiones y resolución de problemas. Nos hemos tenido que enfrentar a varios dilemas como, por ejemplo, a la hora de decidir si usábamos el estado “RUNNING” para el proceso que se está ejecutando (al final hemos decidido no usarlo ya que el proceso en ejecución es el que almacena la variable “running”). Otra toma de decisión se ha dado al debatir si era necesario usar o no la función *mythread_gettid()* para obtener el identificador de un proceso, cuando los TCB contienen un atributo que es el propio identificador del proceso al que se accede mediante “proceso.tid”.