DISEÑO DE SISTEMAS OPERATIVOS

Planificación de procesos

Marzo 2020

Pablo Escrivá Gallardo 100348802 - 100348802@alumnos.uc3m.es

Cristian K. Gómez López - 100349207 - 100349207@alumnos.uc3m.es

ÍNDICE DE CONTENIDOS

Introducción

En esta primera práctica del curso de Diseño de Sistemas Operativos de la UC3M, hemos implementado 3 políticas de planificación. Round-Robin, Round-Robin/SJF con prioridades, Round-Robin/SJF con posibles cambios de contexto voluntarios. Para facilitar el entendimiento de nuestro código hemos incluido para cada uno de los apartados un diagrama de estados en el que se explica el flujo de acción de nuestro programa haciendo especial énfasis en los estados en los que nos encontramos en cada momento, así como las funciones o eventos que hacen que se migre a otro estado. Por último al final de cada una de las políticas de planificación hemos incluido una batería de pruebas para demostrar el correcto funcionamiento de nuestro programa.

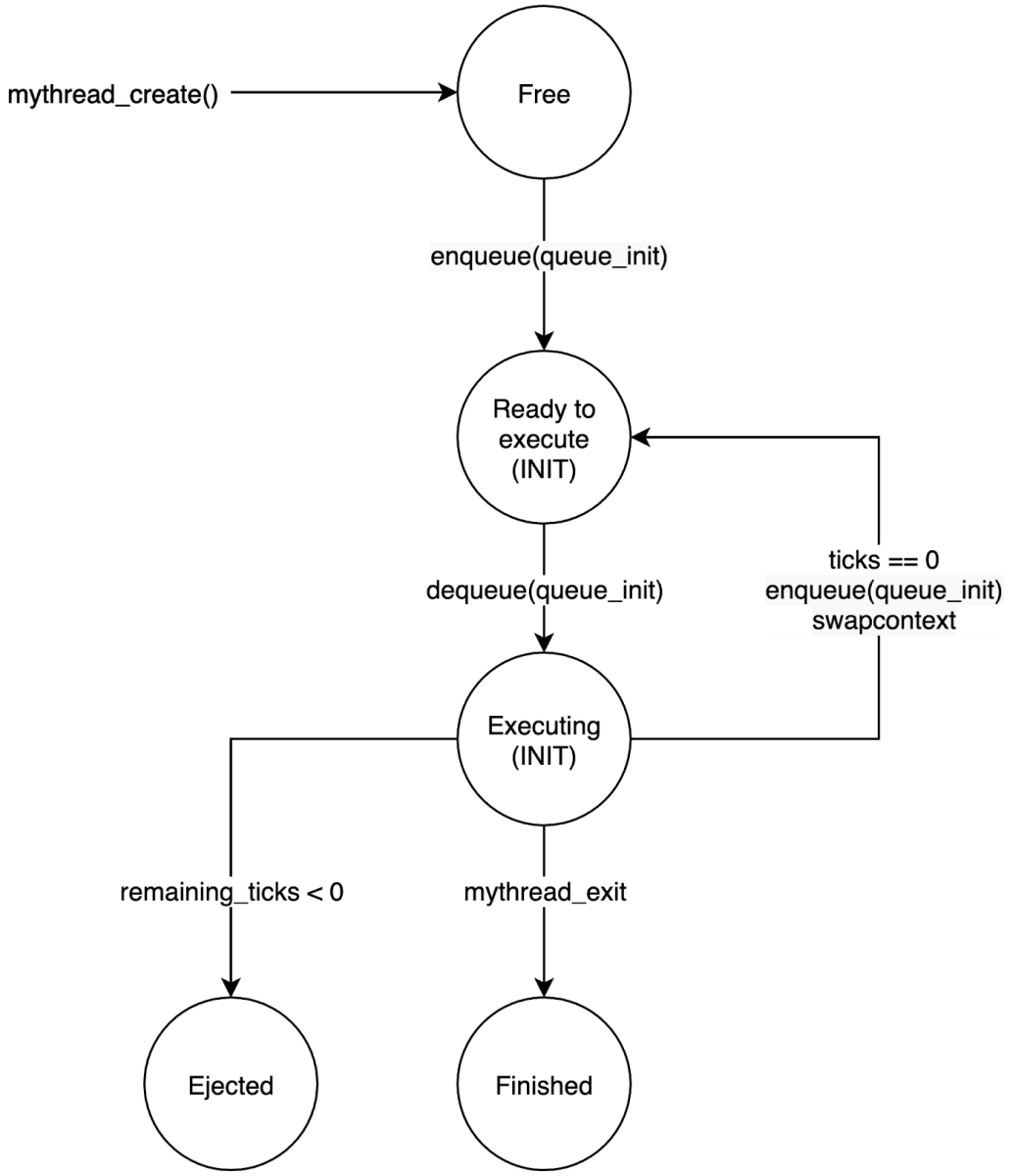
Nota aclaratoria acerca de los apartados de pruebas: Debido a que los algoritmos de planificación de cada apartado se han planteado de manera incremental, las pruebas de cada apartado incluyen también las pruebas de los apartados anteriores. Hacemos esto con intención de mantener la memoria más compacta y facilitar su corrección.

Round Robin

Justificación de las decisiones de diseño

- Hemos utilizado una única cola para procesos listos para ejecución debido a que solo contábamos con un único tipo de prioridad
- Debido a la simplicidad de RR sin prioridades el scheduler ha acabado solamente ocupándose de devolver un proceso desencolado de la lista de procesos listos
- No comprobamos que los procesos se queden sin “remaining ticks”, ya que al no haber prioridades esto queda descartado y así además simplificamos el comportamiento.

Diagrama de estados finitos



Pruebas realizadas

Prueba: RR-01	**Nombre:** Se ejecutan varios procesos idénticos
Descripción: Asignamos a uno de los threads que creamos en main.c un tiempo de ejecución igual esperando que el número de veces que se asigna tiempo de ejecución sea el mismo para todos los creados.	
Resultado Esperado: Todos los threads tienen el mismo número de rodajas y finalizan su ejecución respetando el orden de la cola.	

Prueba: RR-02	**Nombre:** Único proceso en la cola
Descripción: Solamente creamos un proceso en main.c con una duración superior a su tiempo de rodaja.	
Resultado Esperado: No se imprime nada hasta que el proceso termina (es decir, no se realiza un *swapcontext* sobre sí mismo).	

Prueba: RR-03	**Nombre:** Tamaño de rodaja más pequeño
Descripción: Reducimos los ticks por rodaja.	
Resultado Esperado: Se realizan más cambios de contexto que con el tamaño de rodaja original.	

Prueba: RR-04	**Nombre:** Tamaño de rodaja más grande
Descripción: Aumentamos los ticks por rodaja.	
Resultado Esperado: Se realizan menos cambios de contexto que con el tamaño de rodaja original.	

Round Robin/SJF con prioridades

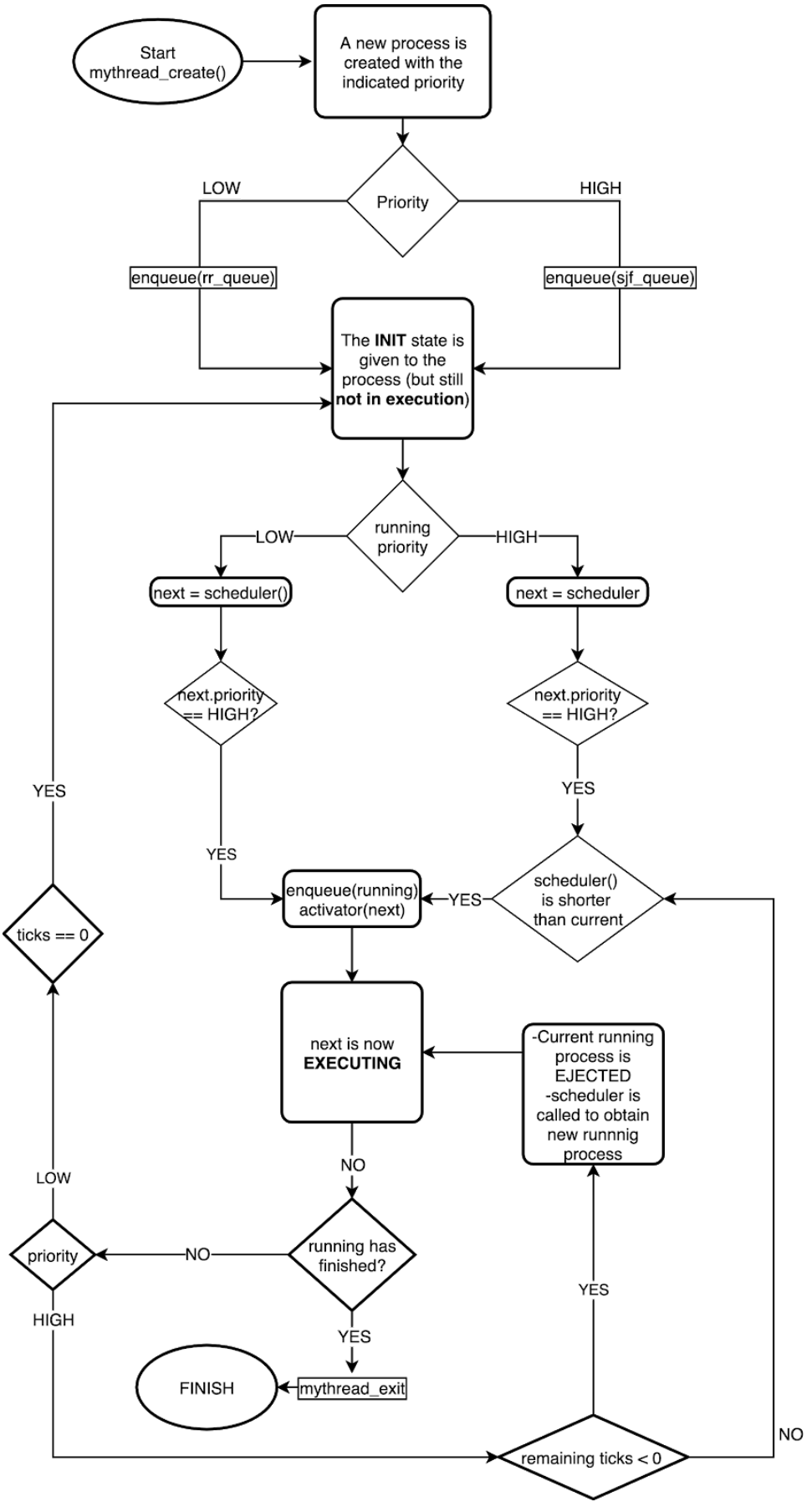
Justificación de las decisiones de diseño

- Hemos usado dos colas para cada tipo de prioridad porque evitaba tener que hacer búsquedas innecesarias si lo comparamos con usar una única cola, teniendo además acceso a la función *sorted_enqueue*, de la que podíamos hacer uso para los procesos de alta prioridad, ordenándolos por el número de “remaining ticks”.
- En un inicio realizamos toda la lógica de cambios de procesos en el manejador de interrupción SW de reloj; no obstante luego decidimos que sería mejor realizar esas operaciones en cada creación de thread porque conseguimos la misma

funcionalidad sin necesitar ninguna función extra de queue.c (como por ejemplo la de inserción al inicio).

- Siguiendo con el punto anterior, vimos que era interesante mover esa lógica previamente en la interrupción SW de reloj, a la función de creación del thread ya que ahí además podíamos realizar la comprobación de si el thread que se creaba era de alta prioridad, estando uno de alta prioridad también ejecutando para así comprobar directamente si el entrante era más corto que el que estaba en ejecución.
- Otra comprobación que decidimos realizar en la creación del thread fue la comprobación de expulsión de un thread de baja prioridad a la entrada de uno de alta.
- En cuanto a la eyección del los threads, al principio pensamos que era interesante realizarlo de alguna manera en la propia función asociada al thread, pero esto era bastante desconcertante ya que la lógica del planificador se salía del propio planificador. Finalmente, y como creímos natural, vimos que era interesante realizar esto en la interrupción de reloj comprobando que los remaining ticks del proceso actual hubiesen bajado de 0.

Diagrama de estados finitos



Pruebas realizadas

Incluye también las pruebas

- Round Robin: RR-01, RR-02, RR-03 y RR-04

Prueba: RRS-01	**Nombre:** Proceso se queda sin remaining_ticks.
Descripción: Asignamos a un proceso de alta que creamos en main.c una función *function_thread_eject* que consume todos sus remaining ticks antes de poder llamar a *mythread_exit.*	
Resultado Esperado: El proceso acaba siendo expulsado del procesador e imprime: *** THREAD 1 EJECTED a continuación realiza un setcontext al siguiente proceso en la cola.	

Prueba: RRS-02	**Nombre:** Proceso de baja prioridad es expulsado por uno de alta.
Descripción: Creamos un proceso de baja prioridad y después uno de alta prioridad.	
Resultado Esperado: El proceso de baja prioridad de expulsado del procesador y se hace swapcontext con el nuevo proceso de alta prioridad, imprimiendo: *** THREAD < baja_prioridad > PREEMTED : SETCONTEXT OF < alta_prioridad >	

Prueba: RRS-03	**Nombre:** Proceso de baja prioridad tiene que esperar.
Descripción: Creamos un proceso de alta prioridad y después uno de baja prioridad.	
Resultado Esperado: El proceso de baja prioridad tiene que esperar a que termine el proceso de alta prioridad para que pueda empezar a ejecutarse.	

Prueba: RRS-04	**Nombre:** Proceso de alta prioridad expulsa a otro de alta prioridad.
Descripción: Creamos un proceso de alta prioridad con una duración elevada y después uno de alta prioridad con poco tiempo restante de ejecución.	
Resultado Esperado: El primer proceso empieza a ejecutarse, pero es expulsado en cuanto el segundo proceso se crea ya que es más corto.	

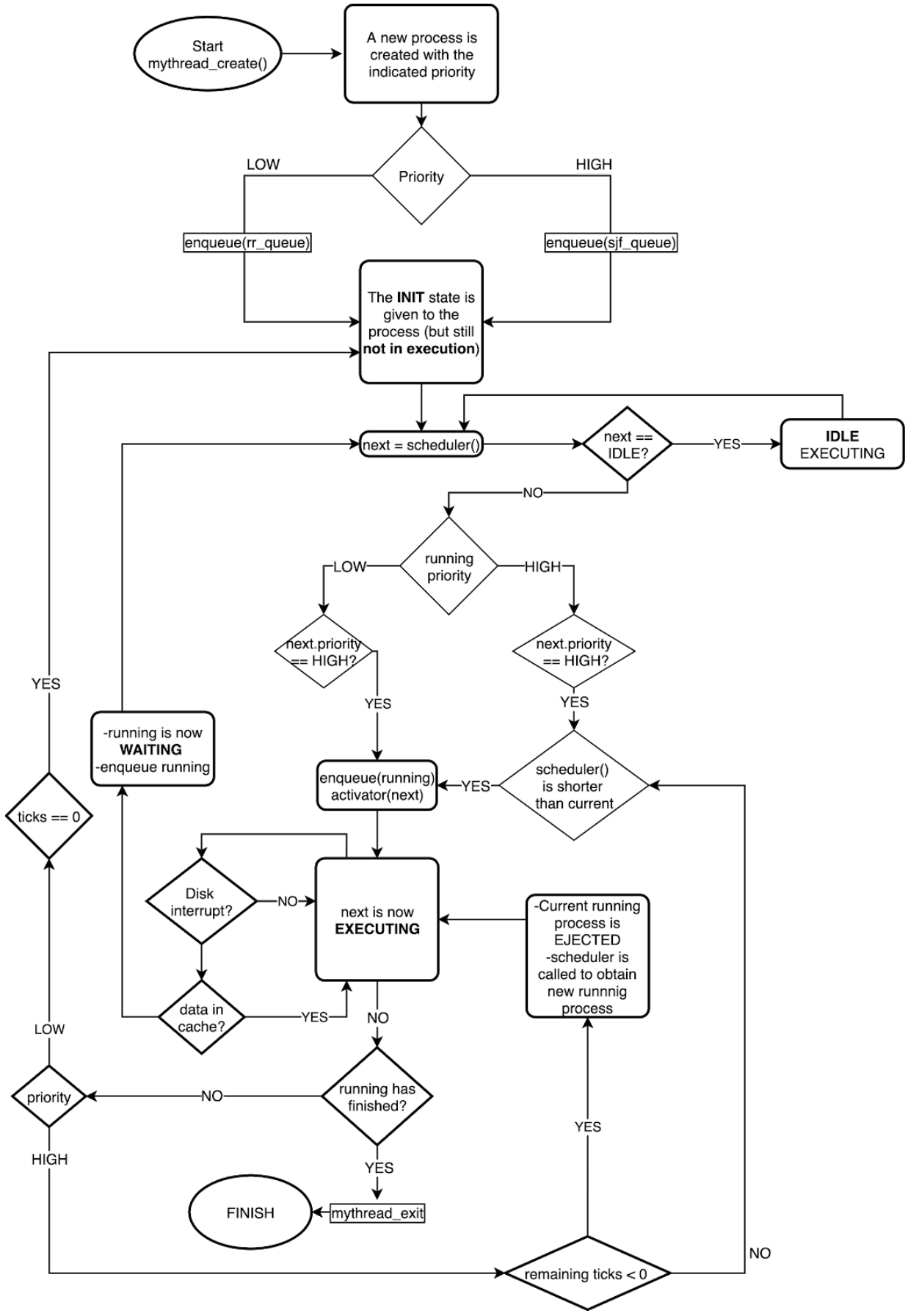
Prueba: RRS-05	**Nombre:** El proceso 0 es de baja prioridad y crea uno de alta prioridad
Descripción: El proceso 0 es de baja prioridad y crea uno de alta prioridad.	
Resultado Esperado: El proceso 0 empieza a ejecutarse, pero deja su ejecución al ser expulsado por el proceso de alta prioridad que acaba de crear. Cuando acaba el proceso de alta prioridad, el proceso 0 puede terminar.	

Round Robin/SJF con posibles cambios de contexto voluntarios

Justificación de las decisiones de diseño

- El lugar de puesta en ejecución(es decir la función que se encargaría de realizar este chequeo y posterior operación) del proceso IDLE fue una de las decisiones más discutidas.
    - En un principio valoramos realizar esto de manera manual en el activator comprobando que el proceso “next” entrante fuese *NULL*, y de esta manera dándole acceso de ejecución al proceso *IDLE*.
    - No obstante nos dimos cuenta de una manera más eficiente que no requería añadirle funcionalidades extra al activator más allá de realizar el cambio de contexto. Pretendíamos delegar el trabajo de devolver un proceso *IDLE* cuando fuese necesario simplemente añadiendo un chequeo más a lo que ya teníamos codificado en el activator en el que simplemente comprobamos después de haber visto que no había ningún tipo de proceso listo, si había alguno bloqueado, si esto era así se estaba dando la condición para ejecutar el proceso *IDLE*(ningún proceso listo y uno o varios procesos bloqueados existentes). Devolvemos entonces bajo esa condición el proceso IDLE sobre el que actuaba luego el activator para realizar el cambio de contexto.
- Hemos decidido no renovar los ticks de reloj de los procesos al ser enviados a la cola de bloqueados(*waiting*), para así respetar de forma estricta el concepto de interrupción. De esta manera cuando se notificaba mediante disk_interrupt que había datos listos para ser leídos, el proceso “recuperaba” el estado que tenía al haber sido interrumpido, incluyendo como decimos también

Diagrama de estados finitos



Pruebas realizadas

Incluye también las pruebas

- Round Robin: RR-01, RR-02, RR-03 y RR-04
- Round Robin SJF: RRS-01, RRS-02, RRS-03, RRS-04 y RRS-05

Prueba: RRSV-01	**Nombre:** Proceso necesita sacar información de disco no presente en caché.
Descripción: Se crea un proceso con una función asociada que incluye una llamada a disco. El dato no se encuentra en caché. Existen otros procesos listos para ejecutarse.	
Resultado Esperado: El proceso pasa a estar en estado WAITING y se introduce en la cola de estados bloqueados. El siguiente proceso en orden de prioridades se pone en ejecución. Cuando el dato está disponible, el proceso que estaba bloqueado pasa a estar en estado INIT y se introduce en su cola correspondiente.	

Prueba: RRSV-02	**Nombre:** Proceso pide un dato que está en caché.
Descripción: Se crea un proceso con una función asociada que incluye una llamada a disco. El dato SI se encuentra en caché.	
Resultado Esperado: El proceso sigue en ejecución ya que no necesita pasar a estado bloqueado.	

Prueba: RRSV-03	**Nombre:** Proceso IDLE entra en ejecución.
Descripción: Se crea un proceso con una función asociada que incluye una llamada a disco. El dato no se encuentra en caché. NO Existen más procesos listos para ejecutarse.	
Resultado Esperado: El proceso pasa a estar en estado WAITING y se introduce en la cola de estados bloqueados. Al no haber más procesos en las colas de procesos listos y además haber al menos un proceso en la cola de procesos bloqueados , el proceso IDLE pasa a entrar a ejecutarse. Cuando el primer proceso recibe el dato, se pone en ejecución dejando al IDLE fuera del procesador.	

Conclusión

Algo sencillo podría ser decir que ha sido pasar a código los ejercicios y pseudocódigo visto en clase; sin embargo, ha habido mucho momentos que el hecho de haber tenido que adaptarnos a un código previamente realizado nos costado bastante ya que muchas funcionalidades no podían ser tocadas, como los comportamientos de las colas, por ejemplo.

Otro aspecto que creemos interesante valorar es el hecho de que hemos invertido bastante tiempo en el entendimiento del código proporcionado. Creemos que tal vez hubiese sido necesaria al menos una clase más para su explicación.

Un aspecto que sí agradecemos es el hecho de que haya sido planteada de manera incremental, ya que esto nos ha permitido avanzar de manera mucho más rápida una vez pudimos tener el Round Robin funcionando.

Creemos además que nos ha dado una visión mucho más detallada del funcionamiento de los algoritmos de planificación aunque entendemos que han sido simulado en modo usuario por motivos didácticos.