Universidad Carlos III de Madrid

Escuela Politécnica Superior (Leganés)

Grado en Ingeniería Informática (3er curso)

Asignatura: Diseño de sistemas operativos

Práctica 1: Planificación de procesos

Elaborado por:

- Estévez Fernández, Andrés Javier
(NIA: 358857 / Grupo 81 /
100358857@alumnos.uc3m.es)

- Zhu, Zhenfeng
(NIA: 363798 / Grupo 81 /
100363798@alumnos.uc3m.es)

Leganés, Madrid. 19 de marzo de 2019.

Índice de contenidos

Introducción

Este documento describe detalladamente la solución implementada para el problema propuesto en esta práctica. También incluye una serie de casos de prueba que tienen como finalidad garantizar el buen funcionamiento del programa. Por último el documento termina con una breve conclusión en la que se habla de los problemas encontrados y las opiniones personales.

Puesto que para implementar la solución se parte del fichero *mythread.c* proporcionado por los profesores y, con el fin de poder diferenciar nuestro código del código base, se decidió aplicar la siguiente medida:

El código de los alumnos está contenido en 2 líneas separadoras:

**********Begin**********
My code.
**********End**********

De esta manera, a la hora de corregir la práctica es más sencillo ubicar el código de los alumnos.

Descripción del código

RR (Round-Robin):

Para implementar el planificador Round Robin se ha creado una cola para guardar los hilos que terminan sus rodajas pero no la ejecución.

Durante la ejecución, el hilo va disminuyendo su número de ticks en *timer_interrupt()*. En cuanto el hilo termine su rodaja y la cola no esté vacía, pasa el siguiente hilo de la cola a la ejecución, y se procede a encolar el hilo que ha terminado su rodaja, en la función *scheduler()*. También, puede ocurrir el caso en el cual el hilo termina su ejecución antes de terminar su rodaja, y en ese caso simplemente pasa el siguiente hilo de la cola a la ejecución y se resetea el estado a FREE, sin tener que encolar el hilo terminado. En ambos casos hay que desactivar las interrupciones para detener el conteo de ticks y resetear la rodaja.

Como se ha comentado anteriormente, en *scheduler()*, mientras que el hilo haya terminado su rodaja pero no su ejecución pasa a la cola. En el caso de que la cola esté vacía y el estado del hilo en ejecución sea FREE, termina el programa imprimiendo el mensaje correspondiente. También se ha considerado el caso de que el hilo no puede cambiar de contexto del mismo si la cola está vacía, para evitar esto se tiene una condición encargada de comprobarlo en *timer_interrupt()*.

Por último, en *scheduler()* se retorna el siguiente hilo a ejecutar de la cola y se pasa a la función *activator(),* y dependiendo de si el hilo haya terminado o termine su rodaje, se realiza un setcontext o swapcontext respectivamente.

RRF (Round-Robin/FIFO con prioridades):

En este programa, además de considerar los requisitos de la parte anterior, hay que tener en cuenta la prioridad de cada hilo. Los hilos de HIGH_PRIORITY serán guardados en la cola de alta prioridad y la ejecución será FIFO, mientras que los hilos de LOW_PRIORITY serán guardados en la cola de baja prioridad con ejecución RR (round-robin).

Para resolver el problema satisfactoriamente se necesitan 2 colas, una para los procesos de alta prioridad y otra para los de baja prioridad. A la hora de crear los hilos en *mythread_create()*, según sea la prioridad de los mismos estos pasarán a la cola de prioridad que le corresponda. También hay que considerar un requisito adicional: si el hilo en ejecución fuera de baja prioridad y el hilo que se ha creado es de alta prioridad, entonces el hilo de baja prioridad será expulsado de la CPU y encolado a la cola de baja prioridad, para que así el hilo recién creado de alta prioridad pase a ejecutarse.

En *timer_interrupt()* y *mythread_exit(),* la implementación es exactamente igual que la parte anterior, aunque *timer_interrupt()* actúa solamente para los hilos de baja prioridad en esta parte.

En *scheduler()* se encola el hilo si es de baja prioridad y ha terminado su rodaja. Luego, se comprueba si quedan hilos en la cola de alta prioridad; en caso afirmativo, se decola el hilo para pasarlo a la función activator, mientras que en caso contrario, se hace lo mismo con la cola de prioridad baja. Es necesario considerar también que si las dos colas están vacías y el estado del hilo en ejecución es FREE, el programa termina.

En *activator()* la implementación es igual que la parte anterior, pero teniendo en cuenta un caso más: si el hilo en ejecución de baja prioridad es expulsado por otro de alta prioridad entonces se imprime el mensaje correspondiente y hace un *swapcontext.*

RRFD (Round-Robin/FIFO con posibles cambios de contexto voluntarios):

En esta tercera parte, el planificador debe permitir realizar un cambio de contexto voluntario mediante la llamada al sistema *read_disk.* El hilo que realiza la llamada *read_disk* deberá liberar la CPU e introducirse en una nueva cola de espera. Para implementar la funcionalidad requerida necesitamos otra cola (denominada cola de bloqueo/espera).

Puesto que para hallar una solución es necesario añadir funcionalidades adicionales basándose en la parte anterior, solamente se explicarán los cambios realizados para añadir estas funcionalidades.

La función *read_disk()* será ejecutada solamente en el caso de que los datos solicitados están en la página de la caché, mediante la función *data_in_page_cache()*, que devuelve un número al azar. En el caso de que los datos no estén en la página, *read_disk()* pone el hilo en ejecución a estado WAITING, lo mete a la lista de espera y pasa el siguiente hilo a ejecutar.

En *disk_interrupt()* se decola un hilo de la cola de espera siempre que no esté vacía. El estado del hilo pasa de WAITING a INIT tras salir de la cola. En caso de que el hilo que ha salido de la cola de espera fuera de alta prioridad y el hilo en ejecución fuera de baja, entonces en *scheduler() se* procede a expulsar el hilo de ejecución para ejecutar el hilo de alta prioridad.

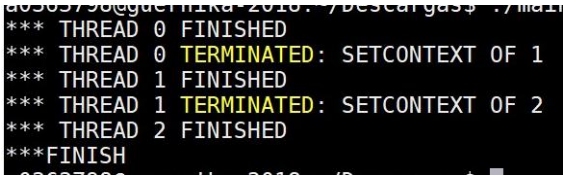
En *scheduler(),* dependiendo de si el estado del hilo en ejecución es INIT o WAITING, será encolado en la cola de baja prioridad o de espera respectivamente. Luego se decola un hilo comprobando primero la cola de alta prioridad; si está vacía, entonces se comprueba la de baja prioridad; si las dos colas están vacías pero no la cola de espera entonces pasa el hilo *idle* a la ejecución. También se considera si las 3 colas están vacías y el estado del hilo en ejecución es FREE, terminando el programa si ocurre la mencionada situación.

Por último, en *activador(),* además de tener los casos de la parte 2, se tiene en cuenta también si el hilo en ejecución es *idle*. En caso afirmativo se imprime el mensaje correspondiente y se realiza setcontext para el siguiente hilo a ejecutar.

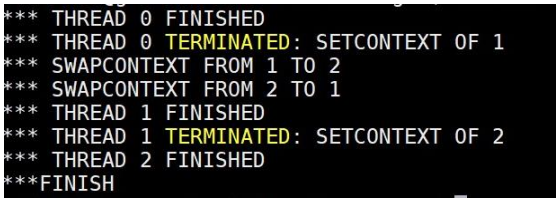
Batería de pruebas

Pruebas RR (Round-Robin):

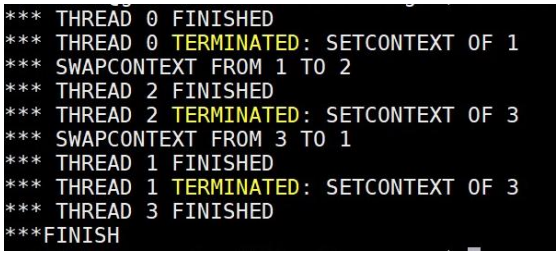
Prueba: Ejecución de solamente hilos sin rodaja.
Output: El hilo 1 y el hilo 2 terminan sus ejecuciones antes de agotar sus ticks



Prueba: Ejecución de solamente hilos con rodaja.
Output: El hilo 1 termina su rodaja, hace el cambio de contexto para el hilo 2, el hilo 2 termina su rodaja, hace el cambio de contexto para el hilo 1, y el hilo 1 y el hilo 2 terminan sus ejecuciones:



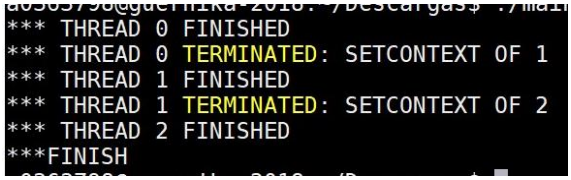
Prueba: Ejecución de un hilo sin rodaja entre 2 hilos con rodaja.
Output: El hilo 1 termina su rodaja, hace el cambio de contexto para el hilo 2, el hilo 2 termina su ejecución, hace el setcontext para el hilo 3. El hilo 3 termina su rodaja, hace el cambio de contexto para el hilo 1. Por último, el hilo 1 y el hilo 3 terminan sus ejecuciones.



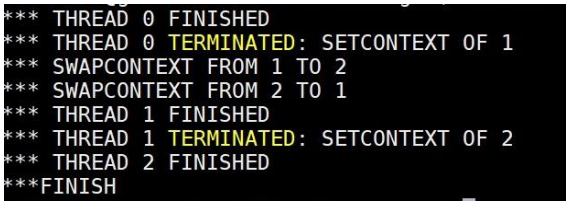
Pruebas RRF (Round-Robin/FIFO con prioridades):

En primer lugar se probaron los casos de prueba de la parte 1 con solamente hilos de prioridad baja, con el fin de comprobar que las funcionalidades de la parte 1 también funcionan correctamente.

Prueba: Ejecución de solamente hilos sin rodaja.
Output: El hilo 1 y el hilo 2 terminan sus ejecuciones antes de agotar sus ticks

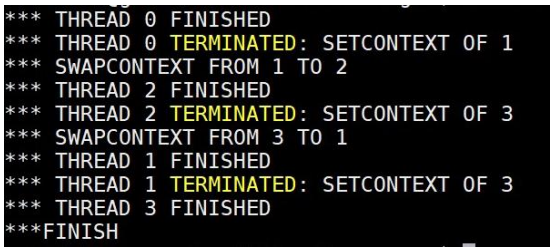


Prueba: Probamos ejecutar solamente hilos con rodaja.
Output: El hilo 1 termina su rodaja, hace el cambio de contexto para el hilo 2, el hilo 2 termina su rodaja, hace el cambio de contexto para el hilo 1, y finalmente ambos hilos sus ejecuciones:

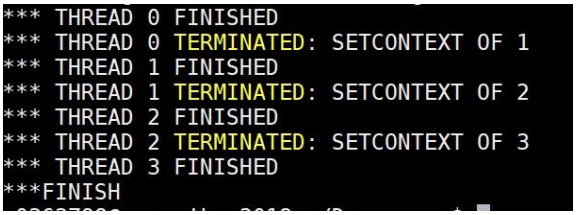


Prueba: Ejecución de un hilo sin rodaja entre 2 hilos con rodaja.
Output: El hilo 1 termina su rodaja, hace el cambio de contexto para el hilo 2, el hilo 2 termina su ejecución, hace el setcontext para el hilo 3. El hilo 3 termina su

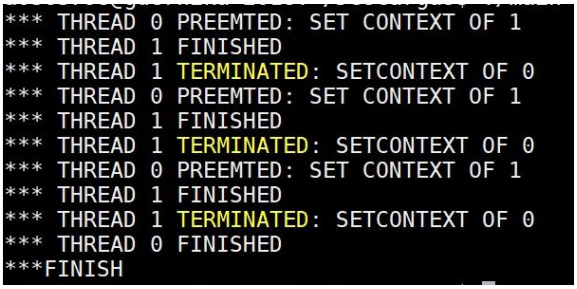
rodaja, hace el cambio de contexto para el hilo 1. Por último, el hilo 1 y el hilo 3 terminan sus ejecuciones.



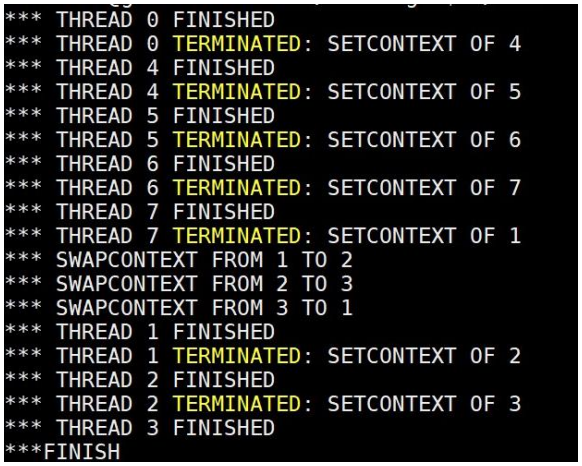
Prueba: Probamos ejecutar hilos de alta prioridad solamente en modo FIFO.
Output: Los hilos ejecutar en el orden llegada hasta terminar su ejecución.



Prueba: Cambio de la prioridad del hilo padre setpriority LOW_PRIORITY y luego ejecutar 3 hilos de alta prioridad.
Output: Se les asignan tid 1 a los 3 hilos de alta prioridad. Esto es porque el hilo 1, debido a que es de alta prioridad, ejecuta hasta terminar su ejecución, y cuando se le asigna el tid al hilo 2 vuelve a asignarle tid al 1, y lo mismo pasa con el hilo 3.



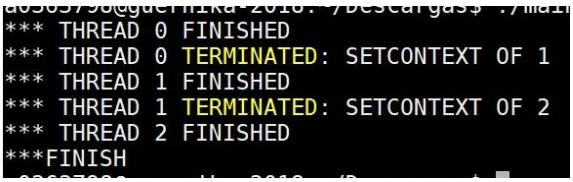
Prueba: Caso original del fichero *main.c* del código base.
Output: En este caso, el primer hilo THREAD 0 ha terminado primero porque es de alta prioridad. Luego pasan los hilos 4, 5, 6 y 7 a la ejecución en modo FIFO debido a que son de alta prioridad. Cuando terminan la ejecución de los hilos de alta prioridad pasan a ejecutarse los hilos de baja prioridad haciendo RR: hilo 1, 2 y 3, hasta terminar la ejecución.



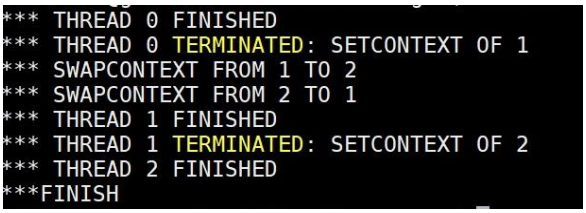
Pruebas RRFD (Round-Robin/FIFO con posibles cambios de contexto voluntarios):

En primer lugar se probaron los casos de prueba de la parte 2 sin read_disk con el fin de comprobar que funcione correctamente.

Prueba: Ejecución de solamente hilos sin rodaja.
Output: El hilo 1 y el hilo 2 terminan sus ejecuciones antes de agotar sus ticks



Prueba: Ejecución de solamente hilos con rodaja.
Output: El hilo 1 termina su rodaja, hace el cambio de contexto para el hilo 2, el hilo 2 termina su rodaja, hace el cambio de contexto para el hilo 1, y el hilo 1 y el hilo 2 terminan sus ejecuciones:



Prueba: Probamos ejecutar un hilo sin rodaja entre 2 hilos con rodaja.
Output: El hilo 1 termina su rodaja, hace el cambio de contexto para el hilo 2, el hilo 2 termina su ejecución, hace el setcontext para el hilo 3. El hilo 3 termina su rodaja, hace el cambio de contexto para el hilo 1. Por último, el hilo 1 y el hilo 3 terminan sus ejecuciones.

```
*** THREAD 0 FINISHED
*** THREAD 0 TERMINATED: SETCONTEXT OF 1
*** SWAPCONTEXT FROM 1 TO 2
*** THREAD 2 FINISHED
*** THREAD 2 TERMINATED: SETCONTEXT OF 3
*** SWAPCONTEXT FROM 3 TO 1
*** THREAD 1 FINISHED
*** THREAD 1 TERMINATED: SETCONTEXT OF 3
*** THREAD 3 FINISHED
***FINISH
```

Prueba: Probamos ejecutar hilos de alta prioridad solamente en modo FIFO.
Output: Los hilos ejecutar en el orden llegada hasta terminar su ejecución.

```
*** THREAD 0 FINISHED
*** THREAD 0 TERMINATED: SETCONTEXT OF 1
*** THREAD 1 FINISHED
*** THREAD 1 TERMINATED: SETCONTEXT OF 2
*** THREAD 2 FINISHED
*** THREAD 2 TERMINATED: SETCONTEXT OF 3
*** THREAD 3 FINISHED
***FINISH
```

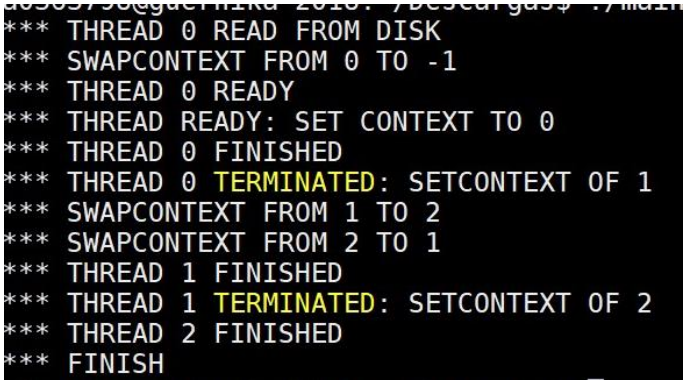
Prueba: Se cambia la prioridad del hilo padre setpriority LOW_PRIORITY y luego se ejecutan 3 hilos de alta prioridad.
Output: En este caso vemos que se les asignan tid 1 a los 3 hilos de alta prioridad, esto es porque el hilo 1, debido a que es de alta prioridad, ejecuta hasta terminar su ejecución, y cuando se le asigna el tid al hilo 2 vuelve a asignarle tid 1, y lo mismo pasa con el hilo 3.

```
*** THREAD 0 PREEMTED: SET CONTEXT OF 1
*** THREAD 1 FINISHED
*** THREAD 1 TERMINATED: SETCONTEXT OF 0
*** THREAD 0 PREEMTED: SET CONTEXT OF 1
*** THREAD 1 FINISHED
*** THREAD 1 TERMINATED: SETCONTEXT OF 0
*** THREAD 0 PREEMTED: SET CONTEXT OF 1
*** THREAD 1 FINISHED
*** THREAD 1 TERMINATED: SETCONTEXT OF 0
*** THREAD 0 FINISHED
***FINISH
```

Prueba: Ejecución de un hilo de alta prioridad expulsado por read_disk y sale de la cola de espera durante la ejecución de un hilo de baja prioridad.
Output: el hilo 1 es expulsado por read_disk, pasa a la cola de espera, y durante la ejecución del hilo 2 sale de la cola de espera imprimiendo THREAD 1 READY, por lo que el hilo 2 es expulsado por ser de baja prioridad, y el hilo 1 termina su ejecución FIFO, luego vuelva a entrar el hilo 2 hasta terminar su ejecución.

```
*** THREAD 0 FINISHED
*** THREAD 0 TERMINATED: SETCONTEXT OF 1
*** THREAD 1 READ FROM DISK
*** SWAPCONTEXT FROM 1 TO 2
*** THREAD 1 READY
*** THREAD 2 PREEMTED: SET CONTEXT OF 1
*** THREAD 1 FINISHED
*** THREAD 1 TERMINATED: SETCONTEXT OF 2
*** THREAD 2 FINISHED
*** FINISH
```

Prueba: Probamos el caso de la ejecución del hilo idle (Invocamos read_disk inmediatamente después de dar alta prioridad al hilo padre):
Output: Se observa que como no hay hilos en las colas de prioridades pasa a la ejecución el hilo idle con tid -1. Luego se produce un disk_interrrupt por lo que el hilo 0 sale de la cola de espera hasta terminar la ejecución. Luego pasan a la ejecución 2 hilos de baja prioridad en modo RR.



Conclusiones y comentarios personales

Durante el desarrollo de la práctica hemos encontrado numerosos problemas a resolver, pero la mayoría de ellos los hemos resuelto gracias a las ayudas proporcionadas por parte de los profesores.

Consideramos que la mayor dificulta de la práctica se basó en lograr entender el código base proporcionado, ya que se encuentra apenas comentado.

Por último, en esta práctica hemos aprendido mucho acerca del funcionamiento interno del planificador de procesos, y el uso práctico del cambio de contexto.