Universidad Carlos III de Madrid

Diseño de Sistemas Operativos

Ingeniería Informática

Práctica 1: Planificación de Procesos

Autores:

Daniel Medina García: 100316850@alumnos.uc3m.es
Alejandro Rodríguez Salamanca: 100315874@alumnos.uc3m.es
Diego Vicente Martín: 100317150@alumnos.uc3m.es

5 de marzo de 2016

Índice

1. Descripción del código

1.1. Descripción del Código Inicial

En el código, nos interesa sobre todo el archivo `mythreadlib.c`, en el cual deberemos implementar las funciones que nos pide el enunciado. A continuación analizamos las distintas partes del código inicial de este archivo:

- La declaración del **array de hilos** global `t_state[]`, cada uno en forma de puntero de `TCB`.
- La otra variable global que utilizaremos, `current`, como referencia al índice del hilo actualmente en ejecución.
- `init_mythreadlib()`: la función en la que se inicializa el hilo padre del resto a ser creados.
- `mythread_create()`: la función que se encarga de crear un hilo ejecutando la función pasada como argumento y con la prioridad también pasada como argumento. Guarda una referencia a su TCB en el primer hueco que encuentre dentro de `t_state[]`.
- `mythread_exit()`: función ejecutado cada vez que uno de los hilos termine. Inicialmente, sólo libera su pila y activa el siguiente proceso.
- Getters y setters de la prioridad, y getter del proceso actualmente en ejecución.
- `timer_interrupt()`: función que se ejecutará en cada interrupción de reloj. Esta función, inicialmente vacía, será de especial importancia en nuestra implementación.
- `scheduler()`: función encargada de seleccionar el siguiente hilo que pasará a ejecución de entre los que se encuentren dispuestos para ello. Inicialmente devuelve el primero que encuentre dentro del array global, y nosotros deberemos implementar aquí los distintos schedulers del enunciado.
- `activator()`: función que cambiará el contexto entre el hilo actual y el siguiente a ejecutar.

Esta implementación inicial no tiene en cuenta prioridades explícitas, pero podría provocar inanición al crear prioridades implícitas en el `scheduler()` al acceder a los elementos del array de forma ordenada empezando siempre por el cero. Este tipo de acceso a `t_state[]` tampoco nos es permitido según el enunciado.

1.2. Descripción del Código de `RR.c`

Es en este archivo en el que se modifica el código inicial para que el planificador siga el algoritmo *Round Robin*. Este algoritmo reparte el tiempo de ejecución de manera equitativa entre todos los procesos listos para su ejecución, de forma que cada determinado tiempo ejecutando (en nuestro caso, indicado por `QUANTUM_TICKS`) produce un cambio involuntario de contexto. Es un algoritmo justo, que evita problemas de inanición al considerar todos los hilos de proceso iguales y seguir un orden *FIFO* dentro de la cola de procesos listos para la elección del siguiente hilo a ejecutar.

Para la implementación de este planificador se deben modificar las variables globales (causando también un pequeño cambio en `init_mythreadlib()`) y las funciones `mythread_create()`, `timer_interrupt()`, `scheduler()` y `activator()`. Además, creamos dos funciones auxiliares (`add_to_queue()` y `get_from_queue()`) que implementarán los accesos seguros a la cola (sin interrupciones de reloj que puedan provocar cambios de contexto mientras se produzcan dichos accesos):

- El planificador utilizará una cola de procesos listos `ready_queue` que contendrá punteros a los TCBs almacenados en el array global mencionado anteriormente para devolver el que la encabece mediante el acceso seguro de `get_from_queue()`. Esta cola se inicializa en `init_mythreadlib()` y se nutrirá de los procesos que dejen de ejecutarse por un CCI cuando se requiera el siguiente a ejecutar en `activator()`.

- La forma de medir cuándo debe un proceso salir de ejecución para dejar paso al siguiente es mediante el atributo `ticks` de su TCB. Es por ello que `mythread_create()` deberá añadir una línea a la inicialización de cada hilo en la que se fije el valor inicial de este campo a `QUANTUM_TICKS`, valor en ticks de cada rodaja de ejecución asignada a cada proceso.

- El anteriormente mencionado `ticks` será reducido en una unidad por cada interrupción de reloj, es decir, en el código de `timer_interrupt()`. Es en esta función donde se comprueba si el proceso ha acabado o no la rodaja y, en caso afirmativo, se pide al planificador un nuevo proceso que posteriormente activará.

- Una vez que el planificador devuelve el proceso que debe ejecutarse, es el `activator()` el que realiza el cambio de contexto. Si se trata de un CCI por finalización, simplemente se fija el contexto del siguiente hilo a ejecutar con `setcontext()`; si, por el contrario, se trata de un CCI, se intercambiarán los contextos del proceso ejecutando y el siguiente mediante `swapcontext()`.

Cabe mencionar también que en `RR.c` quedan modificadas las trazas impresas en consola para adecuarse a las especificadas en el enunciado de la práctica.

1.3. Descripción del Código de `RRF.c`

En `RRF.c` debíamos implementar un planificador que tuviese en cuenta las prioridades de los procesos, ejecutando primero aquellos con alta prioridad (aquellos cuyo atributo `priority` sea igual a `HIGH_PRIORITY`) siguiendo una cola *FIFO* y más tarde aquellos con baja prioridad (cuyo TCB, por el contrario, contenga `LOW_PRIORITY` en `priority`) siguiendo el algoritmo implementado en `RR.c`, *Round Robin*.

Una vez desarrollado `RR.c` no nos fue difícil deducir que necesitábamos dos colas distintas para esta implementación. Las denominamos descriptivamente `high_priority_queue` y `low_priority_queue`, y siguieron el mismo esquema de inicialización que `ready_queue` en `RR.c`. De esta forma, el nuevo `scheduler()` busca antes en la primera cola y en la segunda sólo si la de alta prioridad se encuentra vacía.

Es de especial importancia la modificación realizada a `mythread_create()`. Como ahora el hilo padre (de baja prioridad en principio, aunque podría ser de alta) puede crear un proceso de alta prioridad, éste deberá pasar a ejecución inmediatamente. Se podría implementar esta comprobación en `timer_interrupt()`, pero ello supondría esperar a la siguiente interrupción de reloj. Como tampoco podemos llamar al activador desde otro sitio distinto de `timer_interrupt()`, lo que hacemos aquí es realizar directamente el cambio de contexto en `mythread_create()` como si se tratase del activador.

La interrupción de reloj conllevará distintas acciones según el proceso que se esté ejecutando ahora que tenemos procesos con diferentes prioridades. Así, cuando se esté ejecutando un proceso de alta prioridad, `timer_interrupt()` no realizará ninguna acción más allá de dicha comprobación. Por el contrario, si se trata de un proceso de baja prioridad, se seguirá el mismo esquema que en `RR.c`.

El activador se mantendrá igual que en el apartado anterior, pero cabe destacar la modificación de `add_to_queue()`. Ahora, nuestro método comprueba la prioridad del proceso a encolar para elegir en qué cola debe hacerlo, automáticamente.

1.4. Descripción del Código de `RRFI.c`

En este último apartado debemos eliminar el problema de inanición provocado en `RRF.c` al utilizar un planificador con prioridades estáticas. Para ello, los hilos que lleven mucho tiempo listos esperando por ser de baja prioridad deben ser automáticamente añadidos a la lista de alta prioridad para agilizar su ejecución.

Con el fin de llevar cuenta del tiempo que un proceso lleva esperando en la cola de baja prioridad se utiliza el atributo de su TCB `hungry`:

- Al crear el proceso en `mythread_create()` se inicializará a `STARVING`, que representa el número máximo de interrupciones de reloj que un proceso podrá pasar de seguido en la cola de baja prioridad, `low_priority_queue`.
- Por cada interrupción de reloj, `timer_interrupt()` reducirá en una unidad el campo `hungry` de cada uno de los hilos almacenados en la cola de baja prioridad.
- Si alguno de los hilos llega a tener `hungry` igual a 0, `timer_interrupt()` lo encolará automáticamente a la cola de alta prioridad `low_priority_queue` haciendo uso de `queue_iterate_exchange()`. Esta función accede de forma segura a la cola de baja prioridad recorriéndola buscando algún proceso que deba ser traspasado.

Un proceso que sufrió de inanición y fue pasado a la cola de alta prioridad debe ejecutarse y sólo debe ejecutarse durante una rodaja completa. Es por esto que tenemos que comprobar que si el proceso padre es el que sufrió de inanición y crea un hilo con mayor prioridad que él éste último no toma el contexto. Para ello introducimos `starving_executing`, una variable que tendrá valor 1 sólo cuando un proceso de baja prioridad previamente hambriento se esté ejecutando, y 0 si esto no se cumple. Nuestro `scheduler()` será el encargado de actualizar esta variable cuando sea necesario.

El resto del código se mantendrá igual que en `RRF.c`, con la excepción de que ahora `add_to_queue()` recibe también la cola objetivo como argumento, para poder utilizarla cuando cambiamos un proceso de baja prioridad a la cola de alta.

2. Batería de Pruebas y Análisis de Resultados.

Para asegurar un buen funcionamiento bajo cualquier circunstancia de nuestro código, hemos diseñado 5 test distintos en el archivo `main.c`, que se seleccionan a través de pasarle un atributo al ejecutable para seleccione el test deseado. Si no recibe ningún argumento, el programa elegirá la ejecución por defecto. Los tests elegidos son:

- **Test 0**: En el que probamos una ejecución con **un solo proceso**, el generador. En este test, se inicializa la biblioteca y no se crea ningún otro proceso más. Aunque parezca trivial, es un caso extremo y por tanto hemos decidido incluirlo en la batería de tests.
- **Test 1**: Es la ejecución que realiza el código por defecto, creando procesos con distinta duración y prioridades, separados en el tiempo. Es un caso bastante específico que debería ser bastante representativo del comportameinto de los distintos planificadores.
- **Test 2**: Diseñado para probar el *Round Robin*, se genera un proceso de prioridad alta y otro de prioridad baja, para comprobar que efectivamente el segundo proceso termina antes que el primero. Esta comprobación se repite repetidas veces, con un periodo de espera entre vez y vez.
- **Test 3**: Es la ejecución que realiza el código por defecto pero, esta vez, el proceso cero —el encargado de crear los hilos— tiene prioridad alta. Esto no debería ser relevante en el primer planificador, pero sí en los otros dos, donde se tienen en cuenta las prioridades a la hora de interrumpir la ejecución de un determinado hilo. En este caso el hilo padre no debería perder el contexto al crear un hilo de alta prioridad, contrastando con el Test 0.
- **Test 4**: En el que se generan 4 procesos largos de prioridad baja y, tras ellos, un proceso corto de prioridad alta. Además de ello, la densidad de procesos y su duración así como el hecho de incluir un proceso de alta prioridad hará que los procesos largos sufran inanición.

Con estos 5 tests, pensamos que cubrimos un espectro suficiente amplio a la hora de probar y depurar el código. Al ejecutar estos métodos, tanto desde la terminal como desde un script, podemos redirigir la salida a un fichero para almacenar los resultados. De esta manera, podemos almacener cada uno de los tests de una forma cómoda para comprobar la ejecución entre distintos planificadores.

Haciendo esto podemos observar como Round Robin (`RR.c`) funciona correctamente en el test 0, creando y terminando el hilo generador y acabando ahí la ejecución. En la ejecuciones de los tests 1, 2 y 3 podemos ver como todo se comporta como se espera, ejecutando correctamente las rodajas. No hay ningún detalle realmente destacable entre ellos, ya que Round Robin está ignorando las prioridades y la inanición, así que solo podemos apreciar el comportamiento a la hora de manejar las rodajas y los cambios de contexto.

Por otro lado, las ejecuciones de *Round Robin + FIFO* (`RRF.c`) marcan una clara diferencia con sus análogos de del primer planificador. En el test 1, podemos observar cómo

se expulsa el hilo 0 cada vez que un proceso de alta prioridad se genera. En el test 2 no se aprecian cambios debido a que todos los procesos son de baja prioridad, pero en el test 3 se pueden observar las diferencias de prioridades en la ejecución. En el test 4 también podemos ver cómo se hacen correctamente los cambios de contexto necesarios debido a los procesos de alta prioridad.

Por último, con el planificador *Round Robin* + *FIFO* + Starvation free (`RRFI.c`), podemos apreciar como en los test 1, 3 y especialmente 4 se pueden apreciar como las ejecuciones son iguales a las que podemos encontrar en `RRF.c` pero con los cambios de contexto que han surgido a partir de la inanición de procesos de baja prioridad.

Debido a todas estas pruebas, podemos estar seguros de que el código funciona correctamente y cumple los requisitos pedidos por el enunciado.

3. Conclusiones, Problemas Encontrados y Opiniones Personales

El principal problema al que nos hemos enfrentado ha sido el código inicial: descifrar el funcionamiento de muchos de los métodos proporcionados ha siod una tarea difícil, debido a la naturaleza del código en sí: funciones con aspectos importantes sin comentar, dudas en ciertas partes del funcionamiento, estilo peculiar a la hora de escribir en *C*... Por suerte, una vez pasada la primera impresión y con la ayuda de un poco de prueba y error, fuimos capaces de entender qué estaba sucediendo exactamente en el código.

Otro problema que nos encontramos fue el hecho de ejecutarlo sobre máquinas virtuales. Pensando que todo funcionaría correctamente, las primeras pruebas las hicimos en una máquina virtual de *Ubuntu* corriendo en nuestros ordenadores personales. Sin embargo, pasado un tiempo nos dimos cuenta de que en ciertas ocasiones la salida del programa variaba entre ejecuciones, cosa que después comprobamos que no pasaba ejecutando el mismo código en los ordenadores del laboratorio de informática.