Universidad
Rey Juan Carlos

Minikernel

Ampliación de Sistemas Operativos

Autores:

Sergio Guisado Durán

Ana Hernando Jiménez

Oscar Nydza Nicpoñ

Miguel Santiago Herrero

Índice

Decisiones de diseño

Inclusión de una llamada simple

La implementación de esta función ha sido muy sencilla como se indicaba en el enunciado. No hemos tenido ningún problema respecto a la realización de su diseño ya que la función solo consta de una línea muy básica. Dicha función se ha implementado en el archivo kernel.c además de añadir los cambios necesarios en los respectivos archivos que se indicaban también en el enunciado.

Llamada que bloquea al proceso un plazo de tiempo

Primeramente, añadimos una nueva lista en el archivo kernel.h donde se almacenan los procesos bloqueados, de la misma forma que existe una para procesos listos, y tener constancia de ellos.

Además, hemos añadido una función nueva denominada `cuentaAtrasBloqueados()` para poder actualizar los tiempos de los procesos bloqueados en función de los TICKS de reloj, es decir, poder desbloquearlos eliminándolos de la lista de bloqueados y añadirlos nuevamente a la lista de listos una vez se haya acabado su tiempo de espera.

Por último, la función `dormir()` se encarga de añadir el proceso correspondiente a la lista de bloqueados y eliminarlo de la lista de listos para que cumpla con el tiempo establecido de espera, además de cambiar el contexto y el nivel de interrupción en sus respectivos momentos.

Como en el anterior apartado, se han añadido en los archivos indicados en el enunciado los respectivos cambios para que pudiésemos comprobar, con las diferentes pruebas, la correcta funcionalidad de este apartado.

Mutex

Estructuras de Datos

Para realizar la tarea de creación de funciones relacionadas con el mutex, se han debido de realizar los siguientes cambios en el fichero “kernel.h”. Estos son:

- Modificación de la estructura de datos `BCP_t`. Se han añadido un array de tamaño dado de descriptores que tiene asociados y un valor que guarda cuántos hay actualmente.
- Se han definido las constantes “LIBRE”, “OCUPADO”, “NO_RECURSIVO” y “RECURSIVO” para hacer el código más legible.
- Se ha creado la estructura de datos `mutex`, que guarda el nombre, si está libre u ocupado, si es recursivo o no, el número de procesos bloqueados, una `lista_BCPs` que guarda dichos procesos y el número de bloqueos en caso de ser recursivo.
- Se ha creado `lista_mutex`, que guarda la lista de `mutex` que existen en el sistema.
- Además, se ha creado `lista_espera_mutex`, que guarda los mutex bloqueados debido a que no quedan mutex libres en el sistema.

Explicación de las funciones

En la función `crear_mutex(2)` primero se comprueba la validez del nombre. Para ello, en vez de devolver error si el nombre excede el máximo de caracteres, se sustituye el último por el carácter de "fin de cadena" (`'\0'`). Tras esto se ha creado una función que busca en la lista general de mutex (`lista_mutex`) si el nombre dado existe o no. Si existe se devuelve error y si no, se comprueba si hay algún mutex libre en la lista. Si lo hay, se crea uno y se inserta en dicha posición. Si no, el proceso actual se bloquea y se mete en la `lista_espera_mutex` hasta que se libere alguno.

La operación de creación de mutex, en resumidas cuentas, lo pone como "OCUPADO".

La función `abrir_mutex(1)` comienza comprobando si el proceso actual ha alcanzado el máximo de mutex por proceso. Si es el caso, se lanza error, si no, se busca el mutex en base (reusando la función `buscarMutexPorNombre(1)`) al nombre y si existe, se asigna al proceso actual.

La función `cerrar_mutex(1)` comienza comprobando si el proceso actual tiene guardado el descriptor del mutex que se quiere cerrar. Si es así, se realizan las siguientes comprobaciones:

1. Si lo estoy usando yo, se cierra implícitamente, poniendo a `NULL` el proceso que está usando el mutex.

Independientemente de lo anterior, se elimina al descriptor de la lista de descriptores del proceso actual, se le pone a "LIBRE" y se resta la cuenta total de mutex.

2. Si es recursivo, se liberan todos los procesos que han hecho lock al mutex y están esperando su turno para usarlo. Tras esto se pone el número de bloqueos a cero.
3. Si hay mutex en la lista de espera para crear uno, se saca el primer proceso de dicha lista y se le cambia de estado a "LISTO".

La función `lock(1)` comienza comprobando si dicho mutex existe. Si es el caso, se entra en un bucle que ejecuta hasta que el proceso que esté usando el mutex sea el proceso actual. Esto se consigue bloqueándolo hasta que se cumpla dicha condición. Tras esto, se comprueba si es recursivo y, si no lo es y se ha bloqueado el mutex una vez, se lanza un error. Si no es así, se añade un bloqueo más.

La función `unlock(1)` realiza la función contraria. De nuevo, se comprueba si el mutex existe. También se comprueba si es el proceso actual el que está usándolo, devolviendo error si no es el caso. Se resta un bloqueo y entonces se comprueba si es recursivo o no. Si lo es, se continúa y si no, se termina la ejecución de forma normal. Tras esto, se comprueba si hay algún proceso bloqueando el mutex y si no es el caso, se continúa añadiendo al proceso actual a dicha lista.

Round-Robin

Para la implementación del Round-Robin, hemos añadido dos variables nuevas en el kernel.h. La primera es `ticksPorRodaja` que será la que lleve la cuenta de los ticks que puede estar en ejecución con esta política. La segunda, `procesoAExpulsar` es una variable en la que se irán guardando los procesos una vez hayan consumido toda su rodaja, dejando sitio al siguiente proceso y colocándose en la última posición a esperar su próximo turno en caso de no haber acabado. Estas variables se inicializan en el planificador para que todos los procesos que se vayan lanzando tengan en cuenta estas variables para llevar a cabo la nueva política.

Por otro lado, se ha modificado la interrupción de software, que comprueba si el proceso actual ha utilizado toda su rodaja de tiempo o no. En caso de que si, cede la posición al siguiente proceso y lo lanza.

Por ello, para que interrupción de software sepa si el proceso actual debe ser expulsado o no, se ha añadido a kernel.c la función `roundRobin()` que por cada tick de reloj, decrementa la variable de `ticksPorRodaja` del proceso y comprueba si su valor es `0` o inferior. Si este es el caso, el `procesoAExpulsar` pasa a ser el proceso actual que será colocado al final de la lista de procesos listos por la función de interrupción de software.

Un pequeño añadido para este apartado. Considerando que Round-Robin debe ser cíclico, es decir, que debería haber una ejecución en bucle de los procesos (p1-p2-p3-p4-p5-p1-p2-p3-p4-p5...), hemos modificado el valor de algunos de los parámetros que venían predefinidos en kernel.h. Hemos llegado a esta conclusión puesto que los casos de prueba que debe pasar el Round-Robin son secuenciales y no se ven las interrupciones de software, sino que los procesos terminan uno detrás de otro.

Por esta razón, hemos grabado un pequeño vídeo de una de las pruebas que hemos realizado que muestra las variables que se han modificado para su realización. En el vídeo también se muestra como si que hay interrupciones de software entre los procesos y se van dando paso unos a otros hasta que terminan todos. En const.h se modificaron los valores de las siguientes variables.

- `#define TICK 100` -> En la prueba vale 1000
- `#define TICKS_POR_RODAJA 10` –> En la prueba vale 1

Con estos cambios y modificando el fichero de prueba yosoy.c para que ejecutase un total de 100 iteraciones, se ha conseguido que se vea el objetivo real del Round-Robin.

El vídeo se adjunta en la entrega y se puede visitar desde el siguiente enlace: **https://youtu.be/58qIcmqHd6U**

Terminal

Para la implementación de esta función primero se han declarado en kernel.h la variable de enteros `char_escritos`, que llevará la cuenta de los caracteres que se han introducido por teclado cuando se está o tratando cada carácter o cuando los procesos se encuentran dormidos y las variables de enteros `ind_escribir` e `ind_leer`, que serán el índice una para leer y otra para escribir en el `buffer_char`, siendo el este un array de caracteres, donde se almacenan los caracteres introducidos hasta que se llena (su tamaño es de 8).

Por otro lado se ha realizado la función `leer_caracter()`, que se encuentra en kernel.c, la cual como su nombre indica se encarga de leer cada carácter que se encuentra en el buffer destinado para guardar los valores que se introducen por teclado. Comprueba si hay algo escrito por terminal, tanto si es solo uno como si hay varios dentro del `buffer_char`, por tanto, si hay algo escrito esta función devuelve el valor de cada carácter que haya en el buffer, si no hay nada escrito por terminal o el buffer ya se ha recorrido y no hay ningún valor nuevo el proceso queda bloqueado hasta se introduce un nuevo carácter.

Por último se ha modificado la función `terminal()` que se encuentra en el archivo kernel.c, la modificación consiste en implementar que dicha función consiga tratar los datos que se van introduciendo por la terminal, esta discriminación se hace de la siguiente manera: (cabe recordar que al inicio de la prueba se duermen los dos procesos lectores 3 segundos) se agrega al `buffer_char` los caracteres que entran por teclado hasta que este buffer se llena, después de esto todo carácter que se introduzca no se añadirá al buffer y aparecerá un mensaje informando que se ha llenado. Cuando los procesos se despierten y comiencen a tratar los datos que se encuentran en el buffer este se liberará para que el usuario pueda seguir introduciendo caracteres. La ejecución terminará cuando se lean 10 caracteres por parte de cada uno de los procesos lectores, que son un total de dos.

Problemas existentes durante la ejecución de las pruebas

Durante la ejecución de “prueba_mutex1”, se ha obtenido un comportamiento no esperado. Esto es debido a que, tras terminar la prueba satisfactoriamente y sin lanzar ningún error, la ejecución termina y se vuelve a la shell del sistema host. Debido a que no se muestra ningún error por pantalla, hemos sido incapaces de depurar el código y encontrar la solución. Cabe destacar que durante la realización de la práctica nos hemos encontrado con errores varios que paraban la ejecución del minikernel, pero estos siempre daban algún mensaje (véanse las funciones de tratamiento de excepciones proporcionadas en “kernel.c”, o los mensajes que muestra el kernel cuando hay un pánico) que facilitaba en alguna manera la depuración, señalando vagamente hacia dónde puede haber surgido el problema.

Durante la ejecución de “prueba_mutex2”, nos hemos encontrado con otro problema que, por falta de tiempo, hemos sido incapaces de depurar. Se trata de que, a pesar de que se crean satisfactoriamente tanto el subproceso “mutex1”, como “mutex2”, sólo “mutex1” termina satisfactoriamente (además del proceso “prueba_mutex2”). Eso sabemos con certeza que es debido a que no despierta aún después de quedar libre el mutex “m2”. Sospechamos que puede ser debido a que no se ejecuta la función `unlock(1)` de manera correcta, pero nos resulta extraño ya que no se lanza ningún error adicional que pueda señalar hacia el error que existe.

Feedback

A pesar de lo didáctica que ha sido la realización de esta práctica, tiene fallos que se podrían mejorar. El principal es la dificultad a la hora de depurar el código. Como se ha mencionado anteriormente, ha habido numerosas ocasiones en las que el código se comportaba de una manera inesperada y, en lugar de tener la posibilidad de depurar de manera sencilla, se ha tenido que recurrir al uso de funciones que imprimen por pantalla. Aún así, de no ser por dichas funciones que hemos tenido que añadir a mano, el único feedback que se da es el que se recibe cuando se ejecutan las pruebas, a excepción del que se recibe cuando se produce alguna excepción. Como anécdota se puede decir que en un momento dado, el programa terminaba de manera no esperada y, mediante el uso de las funciones `printk()` y `printf()`, se pudo rastrear el error hasta la función `liberar_imagen(1)`, que pertenece al módulo HAL y, por tanto, fue imposible llegar más a fondo. Tuvimos que rehacer desde cero la función en la que estábamos trabajando para conseguir que continuase ejecutando.