PRÁCTICA 1 – MINIKERNEL
AMPLIACIÓN DE SISTEMAS OPERATIVOS

Descripción de la práctica

La práctica consiste en modificar la versión inicial de minikernel que se entrega como material de apoyo para incluir nuevas funcionalidades y añadir multiprogramación al mismo. Se debe aclarar que, siempre que se mantenga la funcionalidad pedida, el alumno tiene libertad a la hora de diseñar el sistema.

Se deben realizar las modificaciones sobre la versión inicial del sistema que se describen en los siguientes apartados.

(2 puntos) Llamada que bloquea al proceso un plazo de tiempo

Se debe incluir una nueva llamada ("int dormir(unsigned int segundos)") que permita que un proceso pueda quedarse bloqueado un plazo de tiempo. El plazo se especifica en segundos como parámetro de la llamada. La inclusión de esta llamada significará que el sistema pasa a ser multiprogramado, ya que cuando un proceso la invoca pasa al estado bloqueado durante el plazo especificado y se deberá asignar el procesador al proceso elegido por el planificador. Nótese que en el sistema sólo existirán cambios de contexto voluntarios y, por lo tanto, sigue sin ser posible la existencia de llamadas al sistema concurrentes. Sin embargo, dado que la rutina de interrupción del reloj va a manipular listas de BCPs, es necesario revisar el código del sistema para detectar posibles problemas de sincronización en el manejo de estas listas y solventarlos elevando el nivel de interrupción en los fragmentos de código correspondientes. Aunque el alumno pueda implementar esta llamada como considere oportuno, a continuación se sugieren algunas pautas:

- Modificar el BCP para incluir algún campo relacionado con esta llamada.
- Definir una lista de procesos esperando plazos.
- Incluir la llamada que, entre otras labores, debe poner al proceso en estado bloqueado, reajustar las listas de BCPs correspondientes y realizar el cambio de contexto.
- Añadir a la rutina de interrupción la detección de si se cumple el plazo de algún proceso dormido. Si es así, debe cambiarle de estado y reajustar las listas correspondientes.
- Revisar el código del sistema para detectar posibles problemas de sincronización y solucionarlos adecuadamente.

Imitando al modo de trabajo de un sistema operativo real, no se debe usar la función "leer_reloj_CMOS" para implementar "dormir".

(5 puntos) Mutex

Se pretende ofrecer a las aplicaciones un servicio de sincronización basado en mutex. Antes de pasar a describir la funcionalidad que van a tener estos mutex, hay que aclarar que su semántica está ideada para permitir practicar con distintas situaciones que aparecen frecuentemente en la programación de un sistema operativo. Concretamente, se van a implementar dos tipos de mutex:

- Mútex no recursivos: Si un proceso que posee un mutex intenta bloquearlo ("lock") de nuevo, se le devuelve un error, ya que se está produciendo un caso trivial de interbloqueo.
- Mútex recursivos: Un proceso que posee un mutex puede bloquearlo nuevamente todas las veces que quiera. Sin embargo, el mutex sólo quedará liberado cuando el proceso lo desbloquee ("unlock") tantas veces como lo bloqueó.

Las principales características de estos mutex son las siguientes:

- El número de mutex disponibles es fijo (constante "NUM_MUT").
- Cada mutex tiene asociado un nombre que consiste en una cadena de caracteres con un tamaño máximo igual a "MAX_NOM_MUT" (incluyendo el carácter nulo de terminación de la cadena).
- Cada proceso tiene asociado un conjunto de descriptores vinculados con los mutex que está usando (similar al descriptor de fichero de UNIX). El número de descriptores por proceso está limitado a "NUM_MUT_PROC". Si al abrir o crear un mutex, no hay ningún descriptor libre, se devuelve un error.
- Cuando se crea un mutex, el proceso obtiene el descriptor que le permite acceder al mismo. Si ya existe un mutex con ese nombre o no quedan descriptores libres, se devuelve un error. En caso de que no haya error, se debe comprobar si se ha alcanzado el número máximo de mutex en el sistema. Si esto ocurre, se debe bloquear al proceso hasta que se elimine algún mutex. La operación que crea el mutex también lo deja abierto para poder ser usado, devolviendo un descriptor que permita usarlo. Se recibirá como parámetro de qué tipo es el mutex (recursivo o no).
- Para poder usar un mutex ya existente, se debe abrir especificando su nombre. Si no quedan descriptores libres, se produce un error. En caso contrario, el proceso obtiene un descriptor asociado al mismo.
- Las primitivas "lock" y "unlock" tienen el comportamiento convencional:
  - "lock": intenta bloquear el mutex. Si el mutex ya está bloqueado por otro proceso, el proceso que realiza la operación se bloquea. En caso contrario se bloquea el mutex sin bloquear al proceso.
  - "unlock": desbloquea el mutex. Si existen procesos bloqueados en él, se desbloqueará a uno de ellos que será el nuevo proceso que adquiera el mutex. La operación "unlock" sobre un mutex debe ejecutarla el proceso que adquirió con anterioridad el mutex mediante la operación "lock".
- Cuando un proceso no necesita usar un mutex, lo cierra. Si el proceso que cierra el mútex lo tiene bloqueado, habrá que desbloquear implícitamente dicho mutex. Nótese que, en el caso de un mutex recursivo, hay que liberarlo con independencia del nivel de anidamiento que tenga. El mutex se eliminará realmente cuando no haya ningún proceso que lo utilice, o sea, no haya

ningún descriptor asociado al mutex. En el momento de la liberación real, es cuando hay que comprobar si había algún proceso bloqueado esperando para crear un mutex debido a que se había alcanzado el número máximo de mutex en el sistema.
- Cuando un proceso termina, ya sea voluntaria o involuntariamente, el sistema operativo debe cerrar todos los mutex que usaba el proceso.

La interfaz de los servicios de mutex va a ser la siguiente:

```
#define NO_RECURSIVO 0
#define RECURSIVO 1

int crear_mutex(char *nombre, int tipo);
int abrir_mutex(char *nombre);
int lock(unsigned int mutexid);
int unlock(unsigned int mutexid);
int cerrar_mutex(unsigned int mutexid);
```

- *crear_mutex*: Crea el mutex con el nombre y tipo especificados. Devuelve un entero que representa un descriptor para acceder al mutex. En caso de error devuelve un número negativo. Habrá que definir dos constantes, que deberían incluirse tanto en el archivo de cabecera usado por los programas de usuario (“servicios.h”) como en el usado por el sistema operativo (“kernel.h”), para facilitar la especificación del tipo de mutex (*NO_RECURSIVO* y *RECURSIVO*)
- *abrir_mutex*: Devuelve un descriptor asociado a un mutex ya existente o un número negativo en caso de error.
- *lock*: Realiza la típica labor asociada a esta primitiva. En caso de error devuelve un número negativo.
- *unlock*: Realiza la típica labor asociada a esta primitiva. En caso de error devuelve un número negativo.
- *cerrar_mutex*: Cierra el mutex especificado, devolviendo un número negativo en caso de error.

Nótese que todas las primitivas devuelven un número negativo en caso de error. Si lo considera oportuno, el alumno puede codificar el tipo de error usando distintos valores negativos.

Con respecto a la operación de crear un mutex, en ella se puede producir una situación conflictiva típica del código del sistema operativo: Una determinada condición, que se cumplía cuando el proceso se bloqueó puede dejar de hacerlo cuando éste se desbloquee, requeriendo, por tanto, volver a evaluarla.

En el caso planteado, esto puede ocurrir con la existencia de un mutex con el mismo nombre que el que se pretende crear.

Por último, hay que resaltar que el diseño de los mutex debe tratar adecuadamente una situación como la que se especifica a continuación:

- El proceso P1 está bloqueado en “crear_mutex”, ya que se ha alcanzado el número máximo de mutex. En la cola de procesos listos hay dos procesos (P2 y P3).
- P2 realiza una llamada a “cerrar_mutex”, que desbloquea a P1, que pasa al final de la cola de procesos listos.
- P2 termina y pasa a ejecutar el siguiente proceso P3.
- P3 llama a “crear_mutex”: ¿qué ocurre?

Hay que asegurarse de que sólo uno de los dos procesos (P1 o P3) puede crear el mutex, mientras que el otro se deberá quedar bloqueado. Se admiten como correctas las dos posibilidades. Una forma de implementar la alternativa en la que P1 se queda bloqueado y P3 puede crear el mutex es usar un bucle en vez de una sentencia condicional a la hora de bloquearse en “crear_mutex” si no hay un mutex libre.

(3 puntos) Round-Robin

Se va a sustituir el algoritmo de planificación FIFO por *round robin*, donde el tamaño de la rodaja será igual a la constante “TICKS_POR_RODAJA”. Con la inclusión de este algoritmo, aparecen cambios de contexto involuntarios, lo que causa un gran impacto sobre los problemas de sincronización dentro del sistema al poderse ejecutar varias llamadas de forma concurrente.

Para solventar estos problemas, en la práctica no se van a permitir los cambios de contexto involuntarios mientras el proceso está ejecutando la rutina de tratamiento de un dispositivo o una llamada al sistema (se trata de un núcleo no expulsivo). Para lograr este objetivo, la solución planteada se va a basar en el mecanismo de interrupción software de planificación, que en este caso será no expulsivo. Así, en la rutina de tratamiento de la interrupción software se realizará el cambio de contexto del proceso actual pasándolo al final de la cola de listos.

La implementación del *round robin* debe cubrir los siguientes aspectos:

- Al asignar el procesador a un proceso, se le debe conceder siempre una rodaja completa, con independencia de si la rodaja previa la consumió completa o no.
- Si no hay procesos listos en el sistema y el proceso en ejecución está realizando el papel de proceso nulo, no se considerará que el proceso está gastando parte de su rodaja.
- Si un proceso que tiene pendiente un cambio de contexto involuntario se bloquea (o termina) como parte de la ejecución de una llamada, no debe aplicarse dicho cambio ni a ese proceso ni a ningún otro. Dado que el módulo HAL no proporciona una función para desactivar la interrupción software, dentro de la rutina de tratamiento de la misma será necesario asegurarse de que el proceso que está en ejecución es precisamente al que se pretendía expulsar (podría ocurrir que ese proceso a expulsar haya dejado voluntariamente el procesador antes de que comience el tratamiento de la interrupción software de planificación). En caso de no ser el mismo, la rutina terminaría sin realizar ningún trabajo.

Estructura del directorio Minikernel

- “Makefile”. Makefile general del entorno. Invoca a los ficheros Makefile de los subdirectorios subyacentes.
- “boot”. Este directorio está relacionado con la carga del sistema operativo. Contiene:
  - “boot”. Programa de arranque del sistema operativo.
- “minikernel”. Este directorio contiene todos los ficheros necesarios para generar el sistema operativo:
  - “Makefile”. Permite compilar el sistema operativo.
  - “kernel”. Fichero que contiene el ejecutable del sistema operativo.
  - “HAL.o”. Fichero objeto que contiene las funciones del módulo HAL. Realmente, es un enlace simbólico al fichero objeto que corresponde con la versión de Linux en la que se está desarrollando la práctica (“HAL.o.old” para versiones más antiguas, y “HAL.o.new” para versiones más modernas).
  - “kernel.c”. Fichero que contiene la funcionalidad del sistema operativo. Este fichero **DEBE SER MODIFICADO** por el alumno para incluir la funcionalidad pedida en el enunciado.
  - “include”. Subdirectorio que contiene los ficheros de cabecera usados por el entorno:
    - “HAL.h”. Fichero que contiene los prototipos de las funciones del HAL.
    - “const.h”. Fichero que contiene algunas constantes útiles.
    - “llamsis.h”. Fichero que contiene los códigos numéricos asignados a cada llamada al sistema. **DEBE SER MODIFICADO** por el alumno para incluir nuevas llamadas.
    - “kernel.h”. Contiene definiciones usadas por “kernel.c” como la del BCP. **DEBE SER MODIFICADO** por el alumno (p. ej. para añadir nuevos campos al BCP).
- “usuario”. Este directorio contiene diversos programas de usuario.
  - “Makefile”. Permite compilar los programas de usuario.
  - “init.c”. Primer programa que ejecuta el sistema operativo. **El alumno puede modificarlo** a su conveniencia para que éste invoque los programas que se consideren oportunos.
  - “*.c”. Programas de prueba. **El alumno puede modificar a su gusto** los ya existentes o incluir nuevos.
  - “include”. Subdirectorio que contiene los ficheros de cabecera usados por los programas de usuario:
    - “servicios.h”. Fichero que contiene los prototipos de las funciones que sirven de interfaz a las llamadas al sistema. **DEBE SER MODIFICADO** por el alumno para incluir la interfaz a las nuevas llamadas.
  - “lib”. Este directorio contiene los programas que permiten generar la biblioteca que utilizan los programas de usuario. Su contenido es:
    - “Makefile”. Compila la biblioteca.
    - “libserv.a”. La biblioteca.
    - “serv.c”. Fichero que contiene la interfaz a los servicios del sistema operativo. Este fichero **DEBE SER MODIFICADO** por el alumno para incluir la interfaz a las nuevas llamadas.
    - “misc.o”. Contiene otras funciones de biblioteca auxiliares.

Pre-evaluación de la práctica

Para que el alumno pueda evaluar que sus modificaciones funcionan, podrá usar las pruebas contenidas en el fichero “init.c”. Para ello, se deben comentar y descomentar las líneas correspondientes.

Prueba del servicio dormir

Se trata de un programa (“prueba_dormir”) que lanza la ejecución de dos procesos que ejecutan el mismo programa (“dormilon”). Este programa invoca dos veces al servicio “dormir”:

- La primera con un valor de un segundo
- La segunda con un valor que depende de su *pid* (“segs=pid+1”).

Se deben comprobar los siguientes aspectos:

- Que los procesos duermen el número de ticks apropiado (100 por cada segundo).
- Que la primera vez que se duermen los dos procesos “dormilon” se despiertan simultáneamente con la misma interrupción de reloj.

Prueba de los mutex

Hay dos pruebas que se detallan a continuación.

Primera prueba

Intenta probar los aspectos relacionados con la creación, apertura y cierre de mutex.

El programa de prueba (“prueba_mutex1”) lanza la ejecución de 4 procesos (“creador1”, “creador2”, “creador3” y “creador4”) que crean cada uno 4 mutex y un quinto proceso (“abridor”) que abre 5 mutex.

El comportamiento de la prueba deberá ser el siguiente:

- La segunda llamada a “crear_mutex” de “creador1” debe dar error ya que existe este mutex.
- Después de crear sus cuatro mutex respectivos, los “creadores” se van a dormir.
- En ese momento debe comenzar a ejecutar el proceso “abridor” que dará un error en el primer “abrir_mutex” ya que no existe y también en el sexto “abrir_mutex”, puesto que ha agotado los descriptores.
- El proceso “abridor” debe quedarse bloqueado en “crear_mutex” ya que se han agotado el número de mutex del sistema.
- Cuando se despiertan los procesos creadores van terminando y liberando implícitamente sus mutex. Cuando “creador1” cierra implícitamente “m1”, debe desbloquearse “abridor” ya que ese mutex desaparece al no estar siendo usado por ningún proceso.
- Cuando “abridor” intente crear por segunda vez “m17” debe devolver un error, puesto que ya existe.

Segunda prueba

Intenta probar los aspectos relacionados con el uso de las primitivas “lock” y “unlock” de los mutex.

El comportamiento de la prueba deberá ser el siguiente:

- El programa “prueba_mutex2” crea un mutex “m1” de tipo no recursivo e intenta hacer un lock sobre un descriptor distinto al devuelto en la creación, por lo que debe de dar un error.
- El programa “prueba_mutex2” crea un mutex “m2” de tipo recursivo y, a continuación, realiza 2 locks sobre cada uno de los mutex creados, el segundo de los cuales, en el caso de “m1” debe fallar.
- Cuando “prueba_mutex2” se va a dormir, los procesos “mutex1” y “mutex2” arrancan pero se quedan bloqueados en los mutex.
- Cuando “prueba_mutex2” despierta, hace un unlock de “m2”, pero al ser recursivo no debe despertar a nadie.
- Cuando “prueba_mutex2” despierta por segunda vez, hace un segundo unlock de “m2”, que debe desbloquear al proceso “mutex1”, el cual entrará a ejecutar cuando “prueba_mutex2” se duerma por tercera vez.
- “mutex1”, una vez obtenido por dos veces el mutex “m2”, se va a dormir quedando todos los procesos bloqueados: “mutex1” durmiendo 2 segundos, “prueba_mutex2” durmiendo 1 segundo y “mutex2” bloqueado en “m1”.
- Cuando “prueba_mutex2” despierta por tercera vez, termina realizándose el cierre implícito de los mutex, que causa el desbloqueo de “mutex2”, que inmediatamente se vuelve a bloquear al intentar hacer el unlock de “m2”.
- “mutex1” se despierta, cierra explícitamente “mutex2”, lo que causa el desbloqueo de “mutex2”, y se duerme por segunda vez.
- “mutex2” ejecuta liberando los mutex y termina.
- Finalmente, “mutex1” se despierta por segunda vez y termina.

Prueba del planificador round_robin

Habrá dos pruebas independientes: una en la que los procesos están continuamente llamando a “printf” (con lo que en la mayoría de los casos la interrupción de reloj que indica el fin de rodaja “pillará” al proceso haciendo una llamada al sistema) y otra en la que los procesos no estarán haciendo llamadas al sistema.

En ambas pruebas se crean 5 procesos y el resultado debe ser tal que los procesos se repartan el tiempo proporcionalmente entre ellos, o sea, que en la salida generada por la prueba, los 5 procesos deben terminar en la fase final de esta salida.

En la primera prueba (“prueba_RR1”) los procesos creados ejecutan el programa “yosoy” que muestra continuamente su identidad por la pantalla.

En la segunda prueba (“prueba_RR2”) los procesos creados ejecutan el programa “mudo” que no escribe por la pantalla pero realiza cálculos aritméticos para “gastar procesador”.

Además de comprobar que las pruebas del *round-robin* funcionan correctamente, asegúrese de que su programa trata adecuadamente las dos siguientes situaciones:

- Cuando un proceso se desbloquea y pasa a ejecutar se le asignará una rodaja completa, no lo que le restaba de la anterior.
- Si se cumple la rodaja mientras un proceso está haciendo una llamada al sistema (p. ej. “lock”), se activa que hay un cambio de contexto involuntario pendiente, pero si cuando el proceso continúa con la llamada se queda bloqueado, se deberá desactivar de alguna forma el cambio pendiente (recuerde que el módulo HAL no proporciona ninguna función para ello) para no repercutirlo a otro proceso.