Práctica Minikernel

Objetivo de la práctica

El objetivo de esta práctica es que el alumno llegue a conocer los principales conceptos relacionados con la gestión de procesos y la multiprogramación. Para ello, el alumno tendrá que escribir el "código del sistema operativo" que realiza la gestión de procesos. Evidentemente, por razones de complejidad, no se trata de código que ejecutará sobre el hardware real, sino que se trabaja en un entorno emulado, denominado *minikernel*, que proporciona un *hardware virtual* sobre el que desarrollar el sistema operativo. A pesar de ello, el alumno se encontrará durante el desarrollo de la práctica con un modo de trabajo y con una problemática similar a la que se presentaría al trabajar escribiendo código "real" de sistema operativo.

La multiprogramación y, en general, la concurrencia son probablemente los temas más importantes en la enseñanza de los sistemas operativos, aunque también son los más complejos de entender. Es muy difícil conseguir comprender lo que ocurre cuando se están ejecutando concurrentemente varias actividades.

Esta dificultad se acentúa notablemente cuando se está trabajando en el nivel más bajo del sistema operativo. Por un lado, en este nivel la mayoría de los eventos son asíncronos. Por otro, en él existe una gran dificultad para la depuración, dado el carácter no determinista del sistema y la falta de herramientas de depuración adecuadas. Todas las consideraciones expuestas hasta ahora explican el motivo de que la programación de sistemas tenga una productividad tan baja y de que los sistemas operativos tengan "mala fama" debido a sus "caídas" y "cuelgues" imprevistos. Por lo tanto, al enfrentar al alumno con esta problemática, es importante resaltar desde el principio las dificultades que se encontrará para la realización de la práctica. Sin embargo, aunque parezca un poco sorprendente, enfrentarse con esos problemas es a su vez un objetivo de la práctica.

Es importante resaltar que en esta práctica se plasman muchos de los conceptos estudiados en la teoría de la asignatura tales como:

- El arranque del sistema operativo.
- El manejo de las interrupciones.
- El manejo de las excepciones.
- El manejo de las llamadas al sistema.
- La multiprogramación.
- El cambio de contexto.
- La planificación de procesos.
- La sincronización entre procesos.
- Los manejadores de entrada/salida.

El entorno de desarrollo de la práctica se puede utilizar en diferentes versiones de Linux, estando preparado para arquitecturas de 32 y de 64 bits.

En el libro de prácticas propuesto en la bibliografía de la asignatura (*Prácticas de Sistemas Operativos: De la base al diseño [http://arcos.inf.uc3m.es/~ssoo-va/ssoo-prac/ssoo-prac.html]*. J. Carretero, F. García y F. Pérez. McGraw-Hill, 2002), se puede encontrar información adicional sobre este entorno de prácticas, aunque no sea estrictamente necesaria para el desarrollo de la práctica.

El entorno de desarrollo

El entorno de desarrollo de la práctica intenta imitar dentro de lo que cabe el comportamiento y estructura de un sistema real. En una primera aproximación, en este entorno se pueden diferenciar tres componentes principales (que se corresponden con los tres subdirectorios que presenta la jerarquía de ficheros del entorno):

- El programa cargador del sistema operativo (directorio "boot").
- El entorno propiamente dicho (directorio "minikernel"), que incluye tanto el *hardware virtual* (módulo "HAL", *Hardware Abstraction Layer*) como el sistema operativo (módulo "kernel").
- Los programas de usuario (directorio "usuario").

A continuación, se describe cada una de estas partes.

Carga del sistema operativo

De forma similar a lo que ocurre en un sistema real, en este entorno existe un programa de arranque que se encarga de cargar el sistema operativo en memoria y pasar el control a su punto de entrada inicial. Este procedimiento imita el modo de arranque de los sistemas operativos reales que se realiza desde un programa cargador. El programa cargador se encuentra en el subdirectorio “boot” y, en un alarde de originalidad, se denomina “boot”. Para arrancar el sistema operativo y con ello el entorno de la práctica, se debe ejecutar dicho programa pasándole como argumento el nombre del fichero que contiene el sistema operativo. Así, suponiendo que el directorio actual corresponde con el subdirectorio “boot” y que el sistema operativo está situado en el directorio “minikernel” y se denomina “kernel”, se debería ejecutar:

```
boot ../minikernel/kernel
```

Nótese que no es necesario ejecutar el programa de arranque desde su directorio. Así, si se está situado en el directorio base de la práctica, se podría usar el siguiente mandato:

```
boot/boot minikernel/kernel
```

Una vez arrancado, el sistema operativo continuará ejecutando mientras haya procesos de usuario que ejecutar. Nótese que este comportamiento es diferente al de un sistema real, donde el administrador tiene que realizar alguna operación explícita para parar la ejecución del sistema operativo. Sin embargo, este modelo de ejecución resulta más conveniente a la hora de probar la práctica.

El módulo HAL

El objetivo principal de este módulo es implementar la capa del hardware y ofrecer servicios que permitan al sistema operativo su manejo. Las principales características de este “hardware” son las siguientes:

- El “procesador” tiene los dos modos de ejecución clásicos: modo privilegiado o núcleo, en el que se ejecuta código del sistema operativo, y modo usuario, que corresponde con la ejecución del código de procesos de usuario.
- El procesador sólo pasa de modo usuario a privilegiado debido a la ocurrencia de algún tipo de interrupción.
- El registro de estado del procesador almacena el modo de ejecución del mismo. Dado que en el retorno del tratamiento de una interrupción, sea del tipo que sea, se restaura el registro de estado, el paso de modo privilegiado a modo usuario se producirá en esta situación de retorno de interrupción cuando en el registro de estado que se restaura el modo es usuario.
- En el sistema hay dos dispositivos de entrada/salida basados en interrupciones: el terminal y el reloj.
- El terminal es de tipo proyectado en memoria. Como se verá en el capítulo de entrada/salida, esto significa que, realmente, el terminal está formado por dos dispositivos independientes: la pantalla y el teclado. La salida de datos a la pantalla del terminal se realiza escribiendo directamente en su memoria de vídeo, no implicando el uso de interrupciones. La entrada de datos mediante el teclado, sin embargo, está dirigida por las interrupciones que se producen cada vez que se pulsa una tecla.
- El reloj temporizador tiene una frecuencia de interrupción programable. Además de este temporizador que interrumpe periódicamente, hay un reloj alimentado con una batería donde se mantiene la hora mientras el equipo está apagado. Este reloj mantiene la hora como el número de milisegundos transcurridos desde el 1 de enero de 1970. Habitualmente, a este reloj se le denomina reloj CMOS.
- El tratamiento de las interrupciones se realiza mediante el uso de una tabla de vectores de interrupción. Hay seis vectores disponibles que corresponden con:
    - Vector 0: Excepción aritmética.
    - Vector 1: Excepción por acceso a memoria inválido.
    - Vector 2: Interrupción de reloj.
    - Vector 3: Interrupción del terminal.
    - Vector 4: Llamada al sistema.
    - Vector 5: Interrupción software.
- Se trata de un procesador con múltiples niveles de interrupción que, de mayor a menor prioridad, son los siguientes:

    - Nivel 3: Interrupción de reloj.
    - Nivel 2: Interrupción del terminal.
    - Nivel 1: Interrupción software y llamada al sistema.
    - Nivel 0: Ejecución en modo usuario.
- En cada momento el procesador ejecuta en un determinado nivel de interrupción, cuyo valor está almacenado en el registro de estado, y sólo admite interrupciones de un nivel superior al actual.
- Inicialmente, el procesador ejecuta en modo privilegiado y en el nivel de interrupción máximo, por lo que todas las interrupciones están inhibidas.
- Cuando el procesador ejecuta en modo usuario (código de los procesos de usuario), ejecuta con un nivel 0, lo que hace que estén habilitadas todas las interrupciones.
- Cuando se produce una interrupción, sea del tipo que sea, el procesador realiza el tratamiento habitual, esto es, almacenar el contador de programa y el registro de estado en la pila, poner el procesador en modo privilegiado, fijar el nivel de interrupción de acuerdo con el tipo de interrupción recibida y cargar en el contador de programa el valor almacenado en el vector correspondiente. Evidentemente, al tratarse de operaciones realizadas por el hardware, todas estas operaciones no son visibles al programador del sistema operativo.
- La finalización de una rutina de interrupción conlleva la ejecución de una instrucción de retorno de interrupción que restaurará el nivel de interrupción y el modo de ejecución previos.
- Se dispone de una instrucción que permite modificar explícitamente el nivel de interrupción del procesador.
- Se trata de un procesador con mapas de entrada/salida y memoria separados. Por tanto, hay que usar instrucciones específicas de entrada/salida para acceder a los puertos de los dispositivos. Hay que aclarar que, realmente, sólo hay un puerto de entrada/salida que corresponde con el registro de datos del teclado, el cual, cuando se produce una interrupción, contiene la tecla pulsada.
- El procesador dispone de seis registros de propósito general de 32 bits, que sólo se tendrán que usar explícitamente para el paso de parámetros en las llamadas al sistema.
- La interrupción software es de tipo no expulsivo y se usará únicamente para la planificación de procesos, concretamente, para la realización de cambios de contexto involuntarios. Téngase en cuenta que se pretende construir un núcleo no expulsivo. Asimismo, obsérvese que, aunque la interrupción software de planificación sea una interrupción software de proceso, al tratarse de un sistema monoprocesador, siempre irá dirigida al proceso en ejecución, por lo que no es necesario especificar el destino de la misma.

Además de incluir funcionalidad relacionada con el hardware, este módulo también proporciona funciones de más alto nivel vinculadas con la gestión de memoria. Con ello, se pretende que el alumno se centre en los aspectos relacionados con la gestión de procesos y la entrada/salida, y no en los aspectos relacionados con la gestión de memoria.

Las funciones ofrecidas por este módulo se pueden clasificar en las siguientes categorías:

- Operaciones vinculadas a la iniciación de los controladores de dispositivos. En su arranque el sistema operativo debe asegurarse de que los controladores de los dispositivos se inician con un estado adecuado. El módulo HAL ofrece una función de este tipo por cada uno de los dos dispositivos existentes. Además, se proporciona un servicio que permite leer la hora del reloj CMOS del sistema, aunque este servicion no se usará en el desarrollo de la práctica.
    - Iniciación del controlador de teclado.

```
void iniciar_cont_teclado();
```

- Iniciación del controlador de reloj. Se especifica como parámetro cuál es la frecuencia de interrupción deseada (número de interrupciones de reloj por segundo).

```
void iniciar_cont_reloj(int ticks_por_seg);
```

- Lectura de la hora almacenada en el reloj CMOS (almacenada como milisegundos transcurridos desde el 1 de enero de 1970). Normalmente, el sistema operativo lee este valor en el arranque, pero luego el mismo se encarga de mantener la hora actualizada. Nótese que el tipo “long long int” permite especificar tipos cuyo tamaño es el doble que el de un entero convencional (por tanto, normalmente 64 bits).

```
unsigned long long int leer_reloj_CMOS();
```

No debe usarse esta función para el desarrollo de la práctica, dado que en ella no se realizan mediciones de tiempo de carácter absoluto.

- Operaciones relacionadas con las interrupciones. En su fase de arranque, el sistema operativo debe iniciar el controlador de interrupciones a un estado válido y deberá instalar en la tabla de manejadores sus rutinas de tratamiento para cada uno de los vectores que hay en el sistema. Asimismo, se proporciona un servicio para cambiar explícitamente el nivel de interrupción del procesador y otro para activar una interrupción software.
    - Iniciación del controlador de interrupciones.

```
void iniciar_cont_int();
```

- Instalación de un manejador de interrupciones. Instala la función manejadora manej en el vector correspondiente a nvector. Existen varias constantes que facilitan la especificación del número de vector.

```
#define EXC_ARITM 0     /* excepción aritmética */
#define EXC_MEM 1       /* excepción en acceso a memoria */
#define INT_RELOJ 2     /* interrupción de reloj */
#define INT_TERMINAL 3  /* interrupción de terminal */
#define LLAM_SIS 4      /* vector usado para llamadas */
#define INT_SW 5        /* vector usado para int. soft. */
void instal_man_int(int nvector, void (*manej)());
```

- Esta función fija el nivel de interrupción del procesador devolviendo el previo. Permite establecer explícitamente un determinado nivel de interrupción, lo que habilita las interrupciones con un nivel superior e inhabilita las que tienen un nivel igual o inferior. Devuelve el nivel de interrupción anterior para así, si se considera oportuno, poder restaurarlo posteriormente usando esta misma función. Están definidas tres constantes que representan los tres niveles de interrupción del sistema.

```
#define NIVEL_1 1 /* Int. Software */
#define NIVEL_2 2 /* Int. Terminal */
#define NIVEL_3 3 /* Int. Reloj */
int fijar_nivel_int(int nivel);
```

- Esta función provoca la activación de una interrupción software, que será tratada cuando el nivel de interrupción del procesador lo posibilite. Nótese que no se proporciona ninguna función para desactivar la interrupción software una vez activada.

```
void activar_int_SW();
```

- Esta función consulta el registro de estado salvado por la interrupción actual y permite conocer si previamente se estaba ejecutando en modo usuario, devolviendo un valor verdadero en tal caso.

```
int viene_de_modo_usuario();
```

- Operaciones de gestión de la información de contexto del proceso. En el contexto del proceso (tipo “contexto_t”), se almacena una copia de los registros del procesador con los valores correspondientes a la última vez que ejecutó este proceso. Se ofrecen funciones para crear el contexto inicial de un nuevo proceso, así como para realizar un cambio de contexto, o sea, salvar el contexto de un proceso y restaurar el de otro.
- Este servicio crea el contexto inicial del proceso estableciendo los valores iniciales de los registros contador de programa (parámetro “pc_inicial”) y puntero de pila, a partir de la dirección inicial de la pila (parámetro “inicio_pila”) y su tamaño (parámetro “tam_pila”). Además, recibe como parámetro una referencia al mapa de memoria del proceso (parámetro “memoria”), que se debe haber creado previamente. Esta función devuelve un contexto iniciado de acuerdo con los valores especificados (parámetro de salida “contexto_ini”). Es importante resaltar que la copia del registro de estado dentro del contexto se inicia con un nivel de interrupción 0 y con un modo de ejecución usuario. De esta forma, cuando el sistema operativo active el proceso por

primera vez mediante un cambio de contexto, el código del proceso se ejecutará en el modo y nivel de interrupción adecuados.

```
void fijar_contexto_ini(
    void *memoria,
    void *inicio_pila,
    int tam_pila,
    void *pc_inicial,
    contexto_t *contexto_ini
);
```

- Esta rutina salva el contexto de un proceso y restaura el de otro. Concretamente, la salvaguarda consiste en copiar el estado actual de los registros del procesador en el parámetro de salida “contexto_a_salvar”. Por su parte, la restauración implica copiar el contexto recibido en el parámetro de entrada “contexto_a_restaurar” en los registros del procesador. Nótese que, al terminar la operación de restauración, se ha “congelado” la ejecución del proceso que invocó esta rutina y se ha “descongelado” la ejecución del proceso restaurado justo por donde se quedó la última vez que ejecutó, ya sea en otra llamada a “cambio_contexto”, si ya ha ejecutado previamente, o desde su contexto inicial, si ésta es la primera vez que ejecuta. El proceso “congelado” no volverá a ejecutar, y, por tanto, no retornará de la llamada a la función de “cambio_contexto”, hasta que otro proceso llame a esta misma rutina especificando como contexto a restaurar el de este proceso. Hay que resaltar que, dado que también se salva y restaura el registro de estado, se recuperará el nivel de interrupción que tenía previamente el proceso restaurado. Si no se especifica el proceso cuyo contexto debe salvarse (primer parámetro nulo), sólo se realiza la restauración.

```
void cambio_contexto(
    contexto_t *contexto_a_salvar,
    contexto_t *contexto_a_restaurar
);
```

- Funciones relacionadas con el mapa de memoria del proceso. Como se ha explicado previamente, el módulo HAL, además de las funciones vinculadas directamente con el hardware, incluye operaciones de alto nivel que gestionan todos los aspectos relacionados con la gestión de memoria. Se ofrecen servicios que permiten realizar operaciones tales como crear el mapa de memoria del proceso a partir del ejecutable y liberarla cuando sea oportuno, así como para crear la pila del proceso y liberarla.
    - Esta rutina crea el mapa de memoria a partir del ejecutable especificado (parámetro “prog”). Para ello, debe procesar el archivo ejecutable y crear las regiones de memoria (código y datos) correspondientes. Devuelve un identificador del mapa de memoria creado, así como la dirección de inicio del programa en el parámetro de salida “dir_ini”.

```
void *crear_imagen(char *prog, void **dir_ini);
```

- Este servicio libera una imagen de memoria previamente creada (parámetro “mem”).

```
void liberar_imagen(void *mem);
```

- Esta rutina reserva una zona de memoria para la región de pila. Se especifica como parámetro el tamaño de la misma, devolviendo como resultado la dirección inicial de la zona reservada.

```
void *crear_pila(int tam);
```

- Este servicio libera una pila previamente creada (parámetro “pila”).

```
void liberar_pila(void *pila);
```

- Operaciones misceláneas. En este apartado se agrupan una serie de funciones de utilidad diversa.
    - Rutinas que permiten leer y escribir, respectivamente, en los registros de propósito general del procesador.

```
long leer_registro(int nreg);
int escribir_registro(int nreg, long valor);
```

- Esta función lee y devuelve un byte del puerto de entrada/salida especificado (parámetro “dir_puerto”). El único puerto disponible en el sistema corresponde con el terminal.

```
#define DIR_TERMINAL 1
char leer_puerto(int dir_puerto);
```

- Ejecuta la instrucción “HALT” del procesador que detiene su ejecución hasta que se active una interrupción.

```
void halt();
```

- Esta función permite escribir en la pantalla los datos especificados en el parámetro “buffer” y cuyo tamaño corresponde con el parámetro “longi”. Para ello, copia en la memoria de vídeo del terminal dichos datos. La rutina de conveniencia “printk” se apoya en la anterior y permite escribir datos con formato, al estilo del clásico “printf” de C.

```
void escribir_ker(char *buffer, unsigned int longi);
int printk(const char *, ...);
```

- Esta función escribe el mensaje especificado (parámetro “mens”) por la pantalla y termina la ejecución del sistema operativo.

```
void panico(char *mens);
```

El módulo "kernel"

Este es el módulo que contiene la funcionalidad del sistema operativo. El alumno recibe como material de apoyo para la realización de la práctica una versión de este módulo que incluye una funcionalidad básica, que deberá modificarse siguiendo las pautas que se detallan posteriormente. A continuación, se describen las principales características generales de esta versión inicial:

- Iniciación. Una vez cargado el sistema operativo, el programa cargador pasa control al punto de entrada del mismo (en este caso, a la función “main” de este módulo). En este momento, el sistema inicia sus estructuras de datos, los dispositivos hardware e instala sus manejadores en la tabla de vectores. En último lugar, crea el proceso inicial “init” y lo activa pasándole el control. Nótese que durante esta fase el procesador ejecuta en modo privilegiado y las interrupciones están inhibidas (nivel de interrupción 3). Sin embargo, cuando se activa el proceso “init” restaurándose su contexto inicial, el procesador pasa a ejecutar en modo usuario y se habilitan automáticamente todas las interrupciones (nivel 0), puesto que en dicho contexto inicial se ha establecido previamente que esto sea así. Obsérvese que, una vez invocada la rutina de cambio de contexto, no se puede volver nunca a esta función ya que no se ha salvado el contexto del flujo actual de ejecución. A partir de ese momento, el sistema operativo sólo se ejecutará cuando se produzca una llamada al sistema, una excepción o una interrupción de un dispositivo.
- Tratamiento de interrupciones externas. Las únicas fuentes externas de interrupciones son el reloj y el terminal. Las rutinas de tratamiento instaladas únicamente muestran un mensaje por la pantalla indicando la ocurrencia del evento. En el caso de la interrupción del teclado, la rutina además usa la función “leer_puerto” para obtener el carácter tecleado. En estas rutinas habrá que incluir progresivamente la funcionalidad pedida en las distintas prácticas.
- Tratamiento de interrupción software. Como ocurre con las interrupciones externas, la rutina de tratamiento sólo muestra un mensaje por la pantalla.
- Tratamiento de excepciones. Las dos posibles excepciones presentes en el sistema tienen un tratamiento común, que depende de en qué modo ejecutaba el procesador antes de producirse la excepción. Si estaba en modo usuario, se termina la ejecución del proceso actual. En caso contrario, se trata de un error del propio sistema operativo. Por tanto, se invoca la rutina “panico” para terminar su ejecución.
- Llamadas al sistema. Existe una única rutina de interrupción para todas las llamadas (rutina “tratar_llamsis”). Tanto el código numérico de la llamada como sus parámetros se pasan mediante registros. Por convención, el

código se pasa en el registro 0 y los parámetros en los siguientes registros (parámetro 1 en registro 1, parámetro 2 en registro 2, y así sucesivamente hasta 5 parámetros). Asimismo, el resultado de la llamada se devuelve en el registro 0. La rutina "tratar_llamsis" obtiene el código numérico de la llamada e invoca indirectamente a través de "tabla_servicios" a la función correspondiente. Esta tabla guarda en cada posición la dirección de la rutina del sistema operativo que lleva a cabo la llamada al sistema cuyo código corresponde con dicha posición.

- En la versión inicial sólo hay tres llamadas disponibles. La función asociada con cada una de ellas está indicada en la posición correspondiente de "tabla_servicios" y, como se ha comentado previamente, será invocada desde "tratar_llamsis" cuando el valor recibido en el registro 0 así lo indique. Las llamadas disponibles en esta versión inicial son las siguientes:
  - Crear proceso. Crea un proceso que ejecuta el programa almacenado en el archivo especificado como parámetro. Esta llamada devolverá un -1 si hay un error y un 0 en caso contrario. Obsérvese que, en este caso, el código que realiza el tratamiento real de la llamada no se ha incluido en la propia función, sino que se ha delegado a una función auxiliar. Puesto que esta rutina auxiliar se invoca de manera convencional, va a recibir los parámetros de la forma habitual.
  - Terminar proceso. Termina la ejecución de un proceso liberando sus recursos.
  - Escribir. Escribe un mensaje por la pantalla haciendo uso de la función "escribir_ker" proporcionada por el módulo HAL. Recibe como parámetros la información que se desea escribir y su longitud. Devuelve siempre un 0.

La versión inicial de este módulo incluye una gestión de procesos básica que corresponde con un sistema monoprogramado. Para ser más precisos, hay que aclarar que en este sistema inicial, aunque se puedan crear y cargar en memoria múltiples programas, el proceso en ejecución continúa hasta que termina, ya sea voluntaria o involuntariamente debido a una excepción. Obsérvese que ninguna de las tres llamadas al sistema disponibles inicialmente puede causar que el proceso pase a un estado de bloqueado. A continuación, se presentan las principales características de la gestión de procesos en este sistema operativo inicial:

- La tabla de procesos ("tabla_procs") es un vector de tamaño fijo de BCPs.
- El BCP dispone de un puntero ("siguiente") que permite que el sistema operativo construya listas de BCPs que tengan alguna relación entre sí. En esta versión inicial sólo aparece una lista de este tipo, la cola de procesos listos ("lista_listos"), que agrupa a todos los procesos listos para ejecutar, incluido el que está ejecutándose actualmente.
- El tipo usado para la cola de listos (tipo "lista_BCPs") permite construir listas con enlace simple, almacenando referencias al primer y último elemento de la lista. Para facilitar su gestión, se ofrecen funciones que permiten eliminar e insertar BCPs en una lista de este tipo. Este tipo puede usarse para otras listas del sistema operativo (por ejemplo, para un mutex). Hay que resaltar que estas funciones están programadas de manera que, cuando se quiere cambiar un BCP de una lista a otra, hay que usar primero la función que elimina el BCP de la lista original y, a continuación, llamar a la rutina que lo inserta en la lista destino. Asimismo, conviene hacer notar que, por simplicidad, el uso de listas basadas en el tipo "lista_BCPs" exige que un BCP no pueda estar en dos listas simultáneamente. Si se quiere plantear un esquema en el que se requiera que un BCP esté en más de una lista, se deberá implementar un esquema de listas alternativo.
- La variable "p_proc_actual" apunta al BCP del proceso en ejecución. Como se comentó previamente, este BCP está incluido en la cola de listos.
- Con respecto a la creación de procesos, la rutina realiza los pasos típicos implicados en la creación de un proceso: buscar una entrada libre, crear el mapa de memoria a partir del ejecutable, reservar la pila del proceso, crear el contexto inicial, rellenar el BCP adecuadamente, poner el proceso como listo para ejecutar e insertarlo al final de la cola de listos.
- La liberación de un proceso cuando ha terminado voluntaria o involuntariamente implica liberar sus recursos (imagen de memoria, pila y BCP), invocar al planificador para que elija otro proceso y hacer un cambio de contexto a ese nuevo proceso. Nótese que, dado que no se va a volver a ejecutar este proceso, se especifica un valor nulo en el primer argumento de "cambio_contexto".
- Por lo que se refiere a la planificación, dado que la versión inicial de este módulo se corresponde con un sistema monoprogramado, el planificador (función "planificador") no se invoca hasta que termina el proceso actual. El algoritmo que sigue el planificador es de tipo FIFO: simplemente selecciona el proceso que esté

primero en la cola de listos. Nótese que, si todos los procesos existentes estuviesen bloqueados (situación imposible en la versión inicial), se invocaría la rutina “espera_int” de la que no se volvería hasta que se produjese una interrupción.

- Hay que resaltar que en este sistema operativo no existe un proceso nulo. Si todos los procesos existentes están bloqueados en un momento dado, lo que no es posible en la versión inicial, es el último en bloquearse el que se queda ejecutando la rutina “espera_int”. Esto puede resultar sorprendente al principio, ya que se da una situación en la que la cola de listos está vacía, pero sigue ejecutando el proceso apuntado por “p_proc_actual”, aunque esté bloqueado. La situación es todavía más chocante cuando termina el proceso actual estando los restantes procesos bloqueados, puesto que en este caso es el proceso que ha terminado el que se queda ejecutando el bucle de la función “planificador” hasta que se desbloquee algún proceso. Obsérvese que alguien tiene que mantener “vivo” al sistema operativo mientras no hay trabajo que hacer.

Los programas de usuario

En el subdirectorio “usuario” existen inicialmente un conjunto de programas de ejemplo que usan los servicios del minikernel. De especial importancia es el programa “init”, puesto que es el primer programa que arranca el sistema operativo. En un sistema real este programa consulta archivos de configuración para arrancar otros programas que, por ejemplo, se encarguen de atender a los usuarios (procesos de “login”). De manera relativamente similar, en nuestro sistema, este proceso hará el papel de lanzador de otros procesos, aunque en nuestro caso no se trata de procesos que atiendan al usuario, puesto que el sistema no proporciona inicialmente servicios para leer del terminal. Se tratará simplemente de programas que realizan una determinada labor y terminan. Como ocurre en un sistema real, los programas tienen acceso a las llamadas al sistema como rutinas de biblioteca. Para ello, existe una biblioteca estática, denominada “libserv.a”, que contiene las funciones de interfaz para las llamadas al sistema.

```
int crear_proceso(char *programa);
```

```
int terminar_proceso();
```

```
int escribir(char *texto, unsigned int longi)
```

Los programas de usuario no deben usar llamadas al sistema operativo nativo, aunque sí podrán usar funciones de la biblioteca estándar de C, como, por ejemplo, “strcpy” o “memcpy”.

La biblioteca “libserv.a” está almacenada en el subdirectorio “usuario/lib” y está compuesta de dos módulos:

- “serv”. Contiene las rutinas de interfaz para las llamadas. Se apoya en una función del módulo “misc” denominada “llamsis”, que es la que realmente ejecuta la instrucción de llamada al sistema. Para hacer accesible a los programas una nueva llamada, el alumno deberá modificar este archivo para incluir la rutina de interfaz correspondiente.
- “misc”. Como intenta indicar su nombre, este módulo contiene un conjunto diverso de funciones de utilidad. Entre ellas, la definición de la función “printf” que, evidentemente, se apoya en la llamada al sistema “escribir”, de la misma manera que el “printf” en un sistema UNIX se apoya en la llamada “write”. Asimismo, proporciona la función de conveniencia “llamsis”. Esta función facilita la invocación de una llamada al sistema ocupándose de la tediosa labor de rellenar los registros con los valores adecuados y provocando, a continuación, el *trap*. A continuación, se muestra la estructura simplificada de esta función para que se pueda comprender mejor su funcionamiento:

```
int llamsis(int llamada, int nargs, ... /* argumentos */) {
    int i;
    escribir_registro(0, llamada);
    for (i=1; nargs; nargs--, i++)
        escribir_registro(i, args[i]);

    trap();
    return leer_registro(0);
}
```

Todos los programas de usuario utilizan el archivo de cabecera “usuario/include/servicios.h” que contiene los prototipos de las funciones de interfaz a las llamadas al sistema, así como el de la función “printf”.

Pasos para la inclusión de una nueva llamada al sistema

Dado que parte de la labor de la práctica es incluir nuevas llamadas al sistema, se ha considerado oportuno incluir en esta sección los pasos típicos que hay que llevar a cabo en este sistema para hacerlo. Suponiendo que el nuevo servicio se denomina “nueva”, estos son los pasos a realizar:

- Incluir en “minikernel/kernel.c” una rutina (que podría denominarse “sis_nueva”) con el código de la nueva llamada.
- Incluir en “tabla_servicios” (fichero “minikernel/include/kernel.h”) la nueva llamada en la última posición de la tabla.
- Modificar el fichero “minikernel/include/llamsis.h” para incrementar el número de llamadas disponibles y asignar el código más alto a la nueva llamada.
- Una vez realizados los pasos anteriores, el sistema operativo ya incluiría el nuevo servicio, pero sólo sería accesible desde los programas usando código ensamblador. Por lo tanto, es necesario modificar la biblioteca de servicios (fichero “usuario/lib/serv.c”) para que proporcione la interfaz para el nuevo servicio. Se debería también modificar el fichero de cabecera que incluyen los programas de usuario (“usuario/include/servicios.h”) para que dispongan del prototipo de la función de interfaz.
- Por último, hay que crear programas de prueba para este nuevo servicio y, evidentemente, modificar “init” para que los invoque. Asimismo, se debería modificar el fichero “Makefile” para facilitar la compilación de este nuevo programa. Evidentemente, si se usan los programas de prueba contenidos en la distribución inicial, no es necesario reealizar este paso.

Problemas de sincronización dentro del sistema operativo

En la versión inicial, el código de las tres llamadas ejecuta todo el tiempo con las interrupciones habilitadas. Al ir añadiendo la funcionalidad pedida puede ser necesario revisar el código para hacer que ciertas zonas de código ejecuten con las interrupciones de un determinado nivel inhibidas.

Gracias al uso del mecanismo de interrupción software y los cambios de contexto involuntarios diferidos, característicos de los núcleos no expulsivos, no va a haber problemas de sincronización entre llamadas concurrentes. Sólo se presentarán problemas durante la ejecución de interrupciones. Concretamente, habrá que analizar los siguientes conflictos:

- El código de cada llamada al sistema (y de la rutina de tratamiento de la interrupción software) con la rutina del terminal y del reloj.
- El código de la rutina de tratamiento de la interrupción del terminal con la rutina del reloj.
- No habría que analizar posibles conflictos durante la ejecución de la rutina de tratamiento de la interrupción del reloj, ya que tiene máxima prioridad.

Descripción de la funcionalidad pedida

Como se comentó previamente, la práctica va a consistir en modificar la versión inicial 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.

Inclusión de una llamada simple

Se debe añadir una nueva llamada (“obtener_id_pr”) que devuelva el identificador del proceso que la invoca. Como puede observarse, se trata de un servicio que realiza un trabajo muy sencillo. Sin embargo, esta primera tarea servirá para familiarizarse con el mecanismo que se usa para incluir una nueva llamada al sistema. El prototipo de la función de interfaz sería el siguiente:

```
int obtener_id_pr()
```

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”.

Contabilidad del uso del procesador por parte de un proceso

Se va a implementar una función “inspirada” en la llamada “times” de POSIX, que tendrá el siguiente prototipo:

```
int tiempos_proceso(struct tiempos_ejec *t_ejec);
```

Siendo el tipo “struct tiempos_ejec”:

```
struct tiempos_ejec {
    int usuario;
    int sistema;
};
```

Esta llamada devuelve el número de interrupciones de reloj que se han producido desde que arrancó el sistema. Además, si recibe como argumento un puntero que no sea nulo, almacena en el espacio apuntado por el mismo cuántas veces en la interrupción de reloj se ha detectado que el proceso estaba ejecutando en modo usuario (campo “usuario”) y cuántas en modo sistema (campo “sistema”). Nótese que la definición del tipo “struct tiempos_ejec” debería estar disponible tanto para el sistema operativo como para las aplicaciones. Por tanto, se debería incluir tanto en el archivo de cabecera usado por los programas de usuario (“servicios.h”) como en el usado por el sistema operativo (“kernel.h”).

Nótese que cuando no hay ningún proceso listo en el sistema, dado que no hay proceso nulo, estará ejecutando el último que se bloqueó. Es importante resaltar que en ese caso no se le debe acumular gasto de procesador a ningún proceso.

Con respecto a los tres tipos de valores de tiempo que puede devolver esta llamada, hay que comentar los siguiente:

- Sobre el valor devuelto como retorno de la llamada, está relacionado con el tiempo real en el sistema. Normalmente no se usa de forma absoluta, sino que se comparan los valores tomados por dos llamadas realizadas en distintos instantes para medir el tiempo transcurrido entre esos dos momentos.
- Por lo que se refiere a los tiempos estimados de ejecución, se trata de un muestreo que se realiza en cada interrupción de reloj y puede proporcionar datos aproximados sobre cuánto tiempo lleva ejecutando un proceso y en qué modo. Nótese que debería usarse la función “viene_de_usuario” proporcionada por el módulo HAL para poder distinguir las “muestras”.

Además, dado que el parámetro recibido es una referencia a una zona de memoria del proceso de usuario, habrá que asegurarse de que, aunque el parámetro sea erróneo, el sistema operativo no se ve afectado por ello cuando intenta

acceder al mismo. Simplemente, abortará la ejecución del proceso que ha hecho la llamada.

Una posible solución es que el sistema operativo indique en una variable global cuando está accediendo a la zona a la que hace referencia el parámetro. De esta manera, en caso de producirse una excepción en modo sistema, si se estaba accediendo a un parámetro, no se produce una situación de *pánico*, sino que se aborta el proceso.

Nótese que este mismo tratamiento habría que hacerlo en toda llamada que use parámetros que correspondan con punteros a zonas de memoria (por ejemplo, en “crear_proceso” o en “escribir”). Sin embargo, a efectos de la evaluación de la práctica, basta con incluirlo en esta llamada.

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.

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 prentendía 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.

Manejo básico de la entrada por teclado

Se pretende implementar un servicio básico de lectura del terminal, programando la función:

```
int leer_caracter()
```

Esta función lee un carácter del terminal y lo devuelve como resultado. Puede parecer sorprendente que la función devuelva un entero en vez de un carácter, pero, por motivos de homogeneidad, se ha seguido el criterio de que todas las llamadas devuelvan un entero. Téngase en cuenta, asimismo, que la clásica función “getchar” de C, en la que se “inspira” esta función, también devuelve un entero.

Las principales caracteriticas de este manejador del teclado son:

- Ofrece un modo de operación orientado a carácter.
- Asociado al terminal existirá un *buffer* con un tamaño de “TAM_BUF_TERM” caracteres, donde se guardan los caracteres tecleados hasta que algún proceso los solicite.
- Si una petición de lectura encuentra que hay datos disponibles, se satisface inmediatamente. En caso contrario, el proceso queda bloqueado esperando la llegada de un carácter.
- La rutina de interrupción del terminal se encarga de introducir en el *buffer* los datos leídos y desbloquear a los procesos cuando sea oportuno. Si al introducir un carácter, se encuentra que el *buffer* está lleno, la rutina ignora el carácter recibido.
- La rutina de interrupción debe encargarse de realizar el eco de los caracteres que recibe.

Como puede observarse, el manejador constituye un ejemplo paradigmático del problema del productor-consumidor:

- La interrupción del terminal “produce” caracteres.
- La llamada al sistema los “consume”.

Hay que tener un especial cuidado a la hora de controlar los posibles problemas de sincronización en la operación del manejador. Téngase en cuenta que la situación es compleja, ya que puede haber partes del código de la llamada “leer_caracter” que requieran que el nivel de interrupción sea igual a 3 (o sea, todas las interrupciones inhibidas), mientras que otros fragmentos pueden requerir sólo nivel 2 (o sea, sólo las interrupciones de terminal inhibidas). Además de los problemas por el uso de variables globales, puede haber problemas de sincronización más sutiles, típicos de los manejadores de los dispositivos. A continuación, se ilustra de forma genérica este tipo de errores suponiendo una hipotética operación de lectura de un dispositivo:

```
leer() {
    .....
    Programar dispositivo
    Bloquear Proceso
    .....
}
```

Si la interrupción del dispositivo llega antes de que el proceso haya podido bloquearse, puede haber un problema de sincronización. Un escenario similar también se produce en la lectura del terminal en esta práctica.

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

- El proceso P1 está bloqueado en “leer_caracter”, ya que no hay caracteres disponibles. En la cola de procesos listos hay dos procesos (P2 y P3).
- Llega una interrupción del terminal 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 “leer_caracter”: ¿qué ocurre?

Hay que asegurarse de que sólo uno de los dos procesos (P1 o P3) puede leer el carácter tecleado, 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 se lleva el carácter es usar un bucle en vez de una sentencia condicional a la hora de bloquearse en “leer_caracter” si no hay caracteres disponibles.

Código fuente de apoyo

Para facilitar la realización de esta práctica se proporciona un código de apoyo del minikernel. Al extraer su contenido desde el directorio “home” de la cuenta del alumno, se crea el directorio donde se debe desarrollar la práctica. Dentro de este directorio se habrán incluido los siguientes ficheros:

- “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 módulo HAL. NO DEBE SER MODIFICADO.
    - “const.h”. Fichero que contiene algunas constantes útiles. NO DEBE SER MODIFICADO.
    - “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.

Recomendaciones generales

Es importante analizar el código de apoyo proporcionado con la práctica ya que será el punto de partida para la realización de la misma. Se deben llegar a entender las relaciones entre los distintos componentes del sistema.

El alumno tiene libertad a la hora de diseñar el sistema siempre que proporcione la funcionalidad pedida.

Las características de la simulación hacen que la utilización del depurador no sea de gran ayuda. Por ello, se recomienda el desarrollo de funciones de depuración que muestren por la pantalla el contenido de diversas estructuras de datos.

Otro aspecto que conviene resaltar es que, debido al esquema de compilación usado en la práctica, puede ocurrir que un error de programación (como, por ejemplo, usar “pintf” en vez de “printf”) aparezca simplemente como un *warning*

en la fase de compilación y montaje. El error como tal no aparecerá hasta que se ejecute el sistema. En resumen, vigile los *warnings* que se producen durante la compilación.

Documentación que se debe entregar

El mandato a ejecutar en la máquina <tt>triqui.fi.upm.es</tt> es:

```
entrega.soa minikernel.2023
```

Este mandato realizará la recolección de los siguientes ficheros:

- “autores” Fichero con los datos del autor:

DNI APELLIDOS NOMBRE MATRÍCULA

- “memoria.txt” Memoria de la práctica. En ella se deben comentar los aspectos del desarrollo de su práctica que considere más relevantes. Asimismo, puede exponer los comentarios personales que considere oportuno.
- “minikernel/kernel.c” Fichero que contiene la funcionalidad del sistema operativo.
- “minikernel/include/llamsis.h” Fichero que contiene el código numérico asignado a cada llamada.
- “minikernel/include/kernel.h” Fichero que contiene definiciones usadas por “kernel.c”.
- “usuario/include/servicios.h” Fichero que contiene los prototipos de las funciones de interfaz a las llamadas.
- “usuario/lib/serv.c” Fichero que contiene las definiciones de las funciones de interfaz a las llamadas.

Evaluación de la práctica

Existe un corrector automático para esta práctica. Las prácticas entregadas serán corregidas cada noche y se le enviará al alumno el resultado de la corrección por correo electrónico a su cuenta de prácticas. El alumno podrá volver a entregar la práctica con las modificaciones que considere oportunas.

Para evaluar la práctica, usaremos pruebas similares a las contenidas en el fichero “init.c”. Para llevar a cabo cada prueba, se deben comentar y descomentar las líneas correspondientes.

En primer lugar, hay que hacer notar que no se trata de pruebas exhaustivas de la funcionalidad pedida en el enunciado, sino que se comprueban ciertos aspectos que se han considerado como básicos.

A continuación, se comentan las diversas pruebas especificando qué comportamiento se espera en cada una de ellas.

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 comprobrar 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 del servicio tiempos_proceso

Se trata de un programa (“prueba_tiempos”) que pasa por cuatro fases en las que se van tomando tiempos usando este servicio:

- La primera corresponde con un bucle que imprime. El resultado debe de ser tal que el proceso pase la mayor parte del tiempo en modo sistema. Además, debe cumplirse que el número de ticks transcurridos (*reales*) debe ser igual a la suma de los de usuario y sistema, ya que en esta fase el proceso no se ha bloqueado.

- La segunda corresponde con un bucle que sólo realiza cálculos. Por tanto, el proceso estará todo el tiempo en modo usuario. Nuevamente, debe cumplirse que el número de ticks transcurridos (*reales*) debe ser igual a la suma de los de usuario y sistema, ya que en esta fase el proceso no se ha bloqueado.
- En la tercera, el proceso se duerme, luego, las estadísticas de tiempo de usuario y de sistema serán prácticamente nulas.
- En la última fase se realiza una llamada con un parámetro erróneo lo que debe causar que se aborte el proceso, pero no el sistema operativo.

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” blqoueado 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 lock 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.

Prueba de lectura del terminal

Esta prueba requiere que el usuario teclee caracteres de manera que se prueben las distintas opciones de sincronización entre la interrupción y la llamada al sistema.

La prueba (“prueba_term”) lanza dos procesos que ejecutan el programa “lector”. Este programa tiene el siguiente esquema:

- Escribe por la pantalla un mensaje solicitando que se pulsen caracteres. El usuario esperará hasta que salga el mensaje de los dos procesos para pulsar caracteres. De esta forma, se está probando que funciona correctamente la situación en la que varios procesos solicitan leer y se deben bloquear debido a que no hay caracteres disponibles. La primera interrupción debe despertar al primer proceso y la segunda al segundo.
- A continuación, los procesos se van a dormir. Mientras tanto, el usuario tecleará caracteres para probar que funciona correctamente la situación en la que no hay ningún proceso esperando datos.
- A partir de ese momento, cada proceso leerá 10 caracteres.

Evidentemente, se debe comprobar que la información leída por los procesos corresponde con lo tecleado y que la ejecución termina después de teclear 22 caracteres.