Práctica 5: Servicios bloqueantes

El objeto de esta práctica es aprender a implementar servicios bloqueantes Un servicio es bloqueante cuando el proceso de usuario que lo invoca tiene que bloquearse. Para ello vamos a extender el servicio de lectura del manejador de dispositivo construido hasta ahora de modo la función `read` se bloquee cuando el dato no está disponible. El guión de práctica es el que sigue:

1. El mensaje de réplica "Dato no disponible"

En la práctica 4 implementamos un dispositivo `/dev/mouse` que proporcionaba el contenido de la variable `caceres`, un entero de valor siempre 5. Observemos la figura 1. El dato siempre estaba disponible para la nueva tarea `mouse_task`, de modo que esta lo copiaba de forma inmediata al proceso de usuario (3) y replicaba en consecuencia al sistema de ficheros (4). Este, a su vez, replicaba al proceso de usuario `raton.c` para desbloquearlo (5). La invocación a `read` desde el proceso de usuario `raton.c` conseguía el dato y no bloqueaba al proceso invocante.

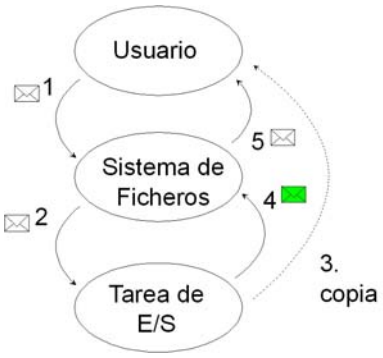

Figura 1. El dato está disponible en el manejador de dispositivo

En muchas ocasiones, sin embargo, el dato que precisa la invocación a `read` no está disponible en la memoria del manejador de dispositivo, de modo que la copia (3) no puede hacerse.

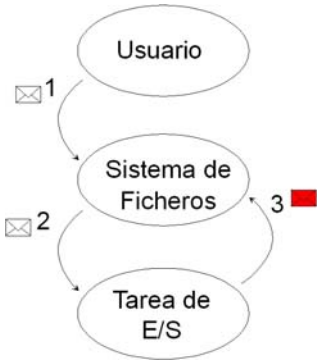

Figura 2. El dato no está disponible en el manejador de dispositivo

Observemos la figura 2. Ahora el sistema de ficheros esta vez no replica al proceso de usuario, de

modo que este permanece bloqueado. En esta ocasión, el mensaje de réplica (3) también es de tipo TASK_REPLY, porque es la réplica al mensaje (2) pero el campo REP_STATUS toma ahora el valor SUSPEND.

Campo	Significado
m_type	Tipo del mensaje. La tarea lo rellena con TASK_REPLY
REP_PROC_NR	La tarea lo rellena con el número del proceso de usuario que invocó read
REP_STATUS	Código de retorno del servicio. Se rellena con **SUSPEND**

Modificación 1.
El resultado de los servicios open y close debe ser OK. Sin embargo, el resultado del servicio read debe ser esta vez SUSPEND en vez de OK. Comprobar que read bloquea indefinidamente al proceso de usuario raton.c.

2. Emisión periódica de un mensaje HARD_INT

Observemos la figura 3. Como la tarea del ratón aún no dispone de una rutina de interrupción (ISR), el mensaje HARD_INT va a ser emitido de forma periódica por la rutina de interrupción del reloj. Recordemos que la rutina de servicio de interrupción de un dispositivo D invoca el procedimiento interrupt(D) para enviar el mensaje HARD_INT a la tarea del dispositivo D.

Figura 3. El dato no está disponible en el manejador de dispositivo

Modificación 2.
La tarea del reloj se encuentra en el fichero fuente /usr/src/kernel/clock.c. La rutina de interrupción es la función clock_handler. El reloj interrumpe 60 veces por segundo, de modo que vamos a introducir en clock.c la nueva variable global segundo, inicializada a cero. Cada activación de clock_handler debe incrementarla en 1 y comprobar si vale 60. En tal caso ha de volver a cero e invocar interrupt(MOUSE).

3. Servicio al mensaje HARD_INT

En este apartado vamos a extender la tarea mouse_task con el servicio al mensaje HARD_INT

Modificación 3.
El nuevo servicio debe incrementar en 1 la variable caceres y mostrarla en pantalla. La información debería aparecer con un periodo aproximado de un segundo. Recordar que HARD_INT no debe ser replicado, porque las rutinas de interrupción no esperan réplica alguna.

Modificación 4.
Una vez comprobado que el servicio a HARD_INT se produce correctamente en tiempo y forma, vamos a sustituirlo por otro. Esta vez va a llevar a cabo los pasos (5) y (6) de la figura 4. El paso (5) será copiar la variable caceres al proceso de usuario tal y como se hizo en la práctica 4. El paso

(6) será enviar al sistema de ficheros un mensaje de tipo `REVIVE` para que reanude al proceso de usuario. ¿Cómo se construye este mensaje? En esta ocasión, el mensaje de réplica (6) también no es de tipo `TASK_REPLY`, porque no es la réplica a ninguna petición, sino de tipo `REVIVE`, y el campo `REP_STATUS` vuelve a tomar ahora el valor `OK`.

Campo	Significado
`m_type`	Tipo del mensaje. La tarea lo rellena con **`REVIVE`**
`REP_PROC_NR`	La tarea lo rellena con el número del proceso de usuario que invocó `read`
`REP_STATUS`	Código de retorno del servicio. Se rellena con **`OK`**

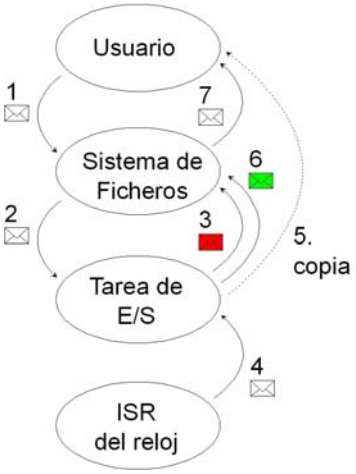

Figura 4. El dato ya está disponible y puede entregarse al proceso de usuario.

4. Un servicio de usuario periódico

El trabajo realizado en `mouse.c` hace que ahora la invocación a `read` suspenda al proceso invocante hasta que llegue el próximo segundo. De lo que se trata ahora es de explotar este comportamiento.

Modificación 5.
`raton.c` invocará `read` en un bucle infinito. Obsérvese que lo que hemos construido es mecanismo que permite a un proceso llevar a cabo una acción periódica con periodo 1 segundo.

```
! cd /usr/home
! mined raton.c
```

```
#include <sys/types.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
main()
{
  int fd;
  int badajoz = 0;
  int cnt     = 0;
  if(0 > (fd = open("/dev/mouse", 0)) {
    perror("Raton: error en open");
    exit(1);
  }

  do {
    if(0 > read(fd, &badajoz, 4)) {
      perror("Raton: error en read");
      exit(1);
    }
    /* ACCIÓN PERIODICA. COMIENZO ------------- */
    if(badajoz != 5) {
      printf("Raton: Algo fue mal: Badajoz = %d\n", badajoz);
      exit(1)
    }
    printf("Raton: Badajoz = %d\n", badajoz);
    /* ACCIÓN PERIODICA. FIN -------------------*/
  } while(cnt++ < 10);

  if(0 > close(fd)) {
    perror("Raton: error en close");
    exit(1);
  }
  printf("Practica 5 superada con exito\n");
}
```

```
! cc raton.c -o raton
! ./raton
```