Diseño de Sistemas Operativos. Junio de 2011.
Ejercicio de procesos y memoria

Se pretende revisar algunos errores de diseño frecuentes en el desarrollo de manejadores de dispositivos. Para ello, se usa como ejemplo simplificado la programación del manejador de teclado para un sistema **monoprocesador** con un núcleo de tipo **no expulsivo** y con ***buffering* de páginas**. Nótese que se trata de un ejemplo similar al incluido en el minikernel, diferenciándose únicamente en que el valor leído se devuelve como parámetro de salida en vez de como valor de retorno, así como por la posible presencia de múltiples dispositivos. Se toma como punto de partida el siguiente código erróneo:

```
struct term_t {
   puerto_t dirp;
   lista_proc cola;
   int n_car;
   char buf[MAX_TAM]; .....
} term[MAX_TERM];

int_teclado(int disp) {
   struct term_t *t = &term[disp];
   char c = leer_puerto(t->dirp);
   if (t->n_car == MAX_TAM) return;
   entrar s.crít. // véase aptdo.(c)
   insertar_en_buf(c,t); //incluye t->n_car++;
   salir s.crít. // véase aptdo.(c)
   if (t->cola->primero != NULL)
     desbloqueo(&t->cola);   }
```

```
leer(int disp, char *p) {
   char c;
   struct term_t *t = &term[disp];
   if (t->n_car == 0)
     bloqueo(&t->cola,t->n_car==0); //véase NOTA
   else {
     entrar s.crít. // véase aptdo.(c)
     c = extraer_de_buf(t); //incluye t->n_car--;
     salir s.crít. // véase aptdo.(c)
     *p = c;
   } }
// NOTA: La macro "bloqueo" recibe como parámetro
// la condición de bloqueo para evitar una
// condición de carrera con la interrupción
// comprobándola antes del bloqueo final.
```

a) Por simplicidad, supóngase inicialmente que se trata de un manejador de acceso exclusivo (es decir, sólo un proceso puede tener abierto el dispositivo y acceder al mismo). Identifique qué error de diseño típico existe en la función `leer` y explique qué parte de la misma hay que eliminar para arreglarlo.

b) Considérese ahora que puede haber múltiples procesos accediendo simultáneamente al mismo dispositivo. Analice primero qué ocurriría en la solución planteada en el apartado anterior cuando se ejecute la siguiente traza:

- P1 lee de un terminal sin datos asociados; Posteriormente, mientras ejecuta P2, de mayor prioridad, llega una interrupción de teclado; Poco después, P2, que no se ha bloqueado en todo este tiempo, realiza la llamada para leer de ese terminal.

En caso de que el comportamiento de la solución planteada en el primer apartado para esta traza sea erróneo, plantee una solución que resuelva este problema, aunque no sea óptima ni asegure un orden FIFO en la lectura.

c) ¿Sería adecuado usar un mecanismo de semáforos para crear la sección crítica requerida por el manejador? ¿Y en caso de un multiprocesador? Cuando no sea adecuado, explique cuál debería utilizarse.

d) Para optimizar el manejador, se plantea eliminar la necesidad de la copia intermedia en caso de que el proceso ya esté esperando por el carácter tecleado. ¿Qué error de diseño típico existe en el siguiente código basado en esta idea?

```
struct term_t {
// mismos campos que versión previa + 1 adicional
   char *dir;
} term[MAX_TERM];

int_teclado(int disp) {
   struct term_t *t = &term[disp];
   char c = leer_puerto(t->dirp);
   if (t->n_car == MAX_TAM) return;
   if (t->cola->primero == NULL) {
     entrar s.crít.
     insertar_en_buf(c,t); //incluye t->n_car++;
     salir s.crít. }
   else {
     *(t->cola->primero->dir) = c;
     desbloqueo(&t->cola);   } }
```

```
leer(int disp, char *p) {
   char c;
   struct term_t *t = &term[disp];
   if (t->n_car == 0) {
     pactual->dir = p;
     bloqueo(&t->cola,t->n_car==0);
   }
   else {
     entrar s.crít.
     c = extraer_de_buf(t); //incluye t->n_car--;
     salir s.crít.
     *p = c;
   }
}
```

e) Detalle la siguiente traza de ejecución, identificando las activaciones del S.O., los cambios de contexto voluntarios e involuntarios, así como qué tipos de fallos de página se producen y cuándo se programa el disco.

- **P** (prioridad alta): solicita una lectura de un terminal sin datos asociados especificando como *buffer* una variable global con valor inicial almacenada en una página a la que no se había accedido previamente. Después de la lectura, **P** aborta la ejecución de **Q** (`kill`) y, posteriormente, termina anormalmente debido a que realiza una división por cero.
- **Q** (prioridad baja): Proceso recién creado.
- La interrupción de teclado se produce estando el proceso **Q** en modo usuario, mientras que la de disco llega estando en la llamada `getpid`. Además, en mitad de la interrupción del disco se activa la del reloj, que es más prioritaria.
- La interrupción de disco y la del reloj usan interrupciones software de sistema para diferir su trabajo menos urgente.

Solución

a) Un error habitual que cometen los programadores noveles de código del sistema operativo a la hora de programar las funciones de un manejador de dispositivo que se activan a través de llamadas al sistema es olvidar especificar las acciones que realiza un proceso después de una operación de bloqueo. Frecuentemente, este tipo de programadores planea, y programa, concienzudamente todas las acciones que realiza el proceso hasta el bloqueo pero, sin embargo, no tiene en cuenta que cuando el proceso se desbloquee continuará su ejecución justo después de la operación de bloqueo y que, por tanto, hay que incluir después de la misma las acciones que correspondan.
En el caso que nos ocupa, cuando un proceso que previamente se ha bloqueado al llamar a `leer` y no disponerse de caracteres, prosigue su ejecución al introducir el usuario un carácter, continuará justo después del bloqueo, no entrando en la rama `else` y, en consecuencia, completándose la función sin devolver el carácter leído. Bastaría con eliminar esa rama `else`, de manera que el proceso que llama a leer realizará siempre la extracción del carácter, tanto si se bloquea como si no lo hace, quedando el siguiente código:

```
leer(int disp, char *p) {
  char c;
  struct term_t *t = &term[disp];

  if (t->n_car == 0)
    bloqueo(&t->cola,t->n_car==0);

  entrar s.crít. // véase aptdo.(c)
  c = extraer_de_buf(t); //incluye t->n_car--;
  salir s.crít. // véase aptdo.(c)
  *p = c;
}
```

b) A continuación, se analiza cómo se desarrollaría la ejecución de la traza planteada usando la solución propuesta en el primer apartado (la que suprimía la rama `else`):

- P1 se bloquea al no haber caracteres disponibles.
- Más adelante, mientras ejecuta P2, se produce la interrupción de teclado que inserta el carácter en el *buffer* y desbloquea a P1, pero al ser este proceso menos prioritario, continúa la ejecución de P2.
- P2 llama a leer y obtiene el único carácter disponible almacenado en el *buffer*.
- Transcurrido un cierto tiempo, P1 ejecuta prosiguiendo su ejecución después de la operación de bloqueo e intenta extraer un carácter del *buffer* cuando no hay ninguno disponible, no devolviendo, por tanto, un carácter válido.

En este escenario, una posible solución consiste en que el proceso P1, al proseguir después del desbloqueo, vuelva a comprobar si hay caracteres disponibles y, en caso negativo, se bloquee nuevamente, todas las veces que sean necesarias. Una forma sencilla de implementar esta funcionalidad es sustituir la sentencia `if` por un bucle `while`:

```
leer(int disp, char *p) {
  char c;
  struct term_t *t = &term[disp];

  while (t->n_car == 0)
    bloqueo(&t->cola,t->n_car==0);

  entrar s.crít. // véase aptdo.(c)
  c = extraer_de_buf(t); //incluye t->n_car--;
  salir s.crít. // véase aptdo.(c)
  *p = c;
}
```

Aunque esta solución puede causar desbloqueos innecesarios y provocar inanición, se usa muy frecuentemente en el diseño de manejadores. Obsérvese la similitud de este esquema con el usado en programas de usuario concurrentes que utilizan variables de condición y envuelven la llamada `cond_wait` en un bucle.

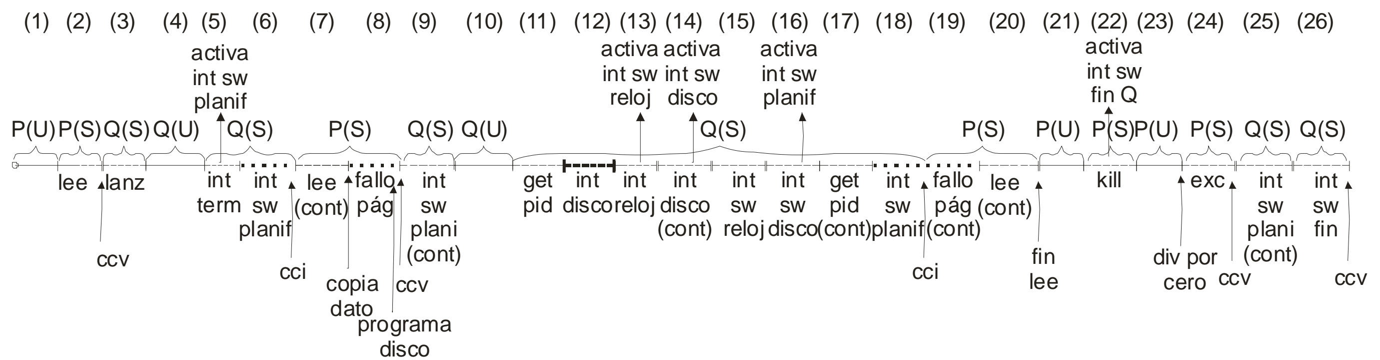
c) La sección crítica especificada en el código pretende resolver las condiciones de carrera entre la función `leer`, invocada en el contexto de una llamada al sistema, y la rutina de tratamiento de la interrupción de teclado. No se pueden, por tanto, usar semáforos para proteger esa sección crítica puesto que no se deben producir bloqueos durante una rutina de tratamiento de interrupción. La solución adecuada es inhibir la interrupción de teclado en el fragmento requerido de la función de lectura.
En el caso de un multiprocesador, tampoco pueden usarse semáforos para resolver esta condición de carrera por los mismos motivos que para un sistema monoprocesador. Sin embargo, no es suficiente únicamente con inhibir la interrupción conflictiva, sino que además hay que usar un mecanismo de espera activa basado en *spinlocks* para asegurar la coherencia (en el código propuesto, se puede declarar una variable de tipo *spinlock* dentro la estructura específica asociada a cada terminal).

d) En el código planteado existe el error que, seguramente, se produce con mayor frecuencia en el contexto de este tipo de programación: el acceso desde una rutina de tratamiento de interrupción al mapa de usuario de un proceso (en este caso, para copiar el carácter tecleado):

```
*(t->cola->primero->dir) = c;
```

El carácter asíncrono de las interrupciones hace que sea imprevisible qué proceso está ejecutando cuando se realiza su tratamiento, por lo que el carácter leído se copiará erróneamente en el mapa de un proceso impredecible, no siendo posible, además, que esté en ejecución el proceso que debería recibir dicho carácter puesto que está bloqueado esperándolo.

e) A continuación, se detalla la traza de ejecución del ejemplo planteado:



A continuación, se detallan cada uno de los eventos que ocurren durante la traza:

1. **P** en modo usuario realiza una llamada al sistema solicitando leer de un terminal, especificando como *buffer* una variable global con valor inicial almacenada en una página a la que no se había accedido previamente.
2. **P** en modo sistema comprueba que no hay caracteres disponibles asociados a ese terminal y se bloquea en la cola de espera asociada al mismo, produciéndose un cambio de contexto voluntario a **Q**.
3. **Q** comienza ejecutando en modo sistema una función de lanzadera que rápidamente se completa pasando a ejecutar en modo usuario la rutina inicial del programa.
4. Durante la ejecución en modo usuario de **Q**, se produce una interrupción del terminal antes referido.
5. La rutina de interrupción del terminal inserta el carácter tecleado en el *buffer* asociado a ese terminal. Dado que el proceso desbloqueado es más prioritario que el actual, se activa una interrupción software expulsiva de planificación para llevar a cabo el cambio de contexto involuntario de **Q** a **P**.
6. La rutina de interrupción software de planificación lleva a cabo el cambio de contexto involuntario requerido, con lo que **P** continúa la ejecución de la llamada al sistema justo en el punto donde quedó detenida.
7. **P** en modo sistema realiza la copia del dato desde el *buffer* del terminal al del usuario especificado en la llamada, pero, dado que no se había accedido previamente a la página que contiene el *buffer* de usuario, se produce un fallo de página que se anida con la llamada al sistema.
8. Al tratarse de una variable global con valor inicial, estará incluida en una región con soporte, por lo que será necesario leerla desde memoria secundaria. Sin embargo, al tratarse de un sistema con *buffering* de páginas, habrá marcos libres disponibles, por lo que no será necesario expulsar una página a memoria secundaria. En consecuencia, habrá que programar el disco para leer la página requerida, produciéndose un cambio de contexto voluntario a **Q**.
9. **Q** reanuda su ejecución en el contexto de una rutina de tratamiento de una interrupción software de planificación donde quedó detenida su ejecución. Al completarla, retorna a modo usuario.
10. **Q** ejecuta en modo usuario hasta que realiza la llamada al sistema `getpid`.
11. Mientras **Q** ejecuta en modo sistema la llamada `getpid`, se produce una interrupción del disco.
12. Durante el tratamiento de la interrupción del disco, se activa una interrupción de reloj, que es más prioritaria en este sistema.
13. El tratamiento de la interrupción del reloj realiza sus labores más urgentes y activa una interrupción software de sistema para gestionar sus operaciones diferidas, terminando su ejecución que retorna a la rutina de interrupción de disco.
14. Prosigue la rutina de interrupción de disco ejecutando sus operaciones más urgentes y activando una interrupción software de sistema para gestionar sus operaciones diferidas. Al completarse, toma control el tratamiento de la interrupción software de sistema asociada al reloj.
15. El tratamiento de la interrupción software de sistema del reloj gestiona sus operaciones diferidas. Cuando se completa, se activa el tratamiento de la interrupción software de sistema del disco.
16. El tratamiento de la interrupción software de sistema del disco, entre otras operaciones no urgentes, desbloquea el proceso **P** y, al ser éste más prioritario, activa una interrupción software de planificación expulsiva para forzar el cambio de contexto involuntario.
17. Al tratarse de un núcleo no expulsivo, **Q** continúa y completa la llamada `getpid`, activándose entonces el tratamiento de la interrupción software de planificación.
18. La rutina de interrupción software de planificación lleva a cabo el cambio de contexto involuntario de **Q** a **P**, con lo que **P** continúa la ejecución de la rutina de tratamiento de fallo de página justo en el punto donde quedó detenida.
19. Se completa la rutina de fallo de página prosiguiendo la ejecución de la llamada.
20. Se completa la llamada al sistema, retornando **P** a modo usuario.
21. **P** ejecuta en modo usuario hasta que realiza la llamada al sistema para abortar el proceso **Q** (`kill`).
22. **P** ejecuta en modo sistema enviando una interrupción software de proceso de terminación dirigida al proceso **Q**, volviendo a modo usuario al completar la llamada.
23. **P** ejecuta en modo usuario hasta que realiza una división por cero, momento en el que se activa la excepción correspondiente del procesador dando paso a su rutina de tratamiento.

24. **P** ejecuta en modo sistema dentro del contexto del tratamiento de la excepción, completando su ejecución y produciéndose un cambio de contexto voluntario a **Q**.
25. **Q** reanuda su ejecución en el contexto de una rutina de tratamiento de una interrupción software de planificación donde quedó detenida su ejecución. Al completarla, se activa el tratamiento de la interrupción software de proceso de terminación.
26. En el tratamiento de la interrupción software de proceso de terminación, **Q** completa su ejecución