Diseño de Sistemas Operativos. Junio de 2014.
Ejercicio de procesos y memoria

La mayoría de los sistemas operativos proporcionan servicios para transferir datos directamente entre dispositivos sin necesidad de que éstos pasen por el espacio de usuario (por ejemplo, la llamada `sendfile` de UNIX). Para analizar las ventajas de este mecanismo, se plantea un fragmento simplificado de pseudo-código del S.O. que corresponde a la implementación de la operación de lectura para un cierto dispositivo de entrada X (llamada `leeX`), la de una operación de escritura para un determinado dispositivo de salida Y (llamada `escribeY`) y la llamada para hacer la transferencia directa entre los dispositivos (`copiaXY`).

```
// pto. entrada llamada al sistema leeX
sis_leeX(char *dir, int tam){
  char *buf = kmalloc(tam);
  do_leeX(buf, tam);
  copy_from_sys_to_user(buf, dir, tam);
  kfree(buf);
}
int tam_pet; char *b; tipo_cola cola_dispX;
do_leeX(char *dirb, int tam){
  tam_pet = tam; b = dirb;
  out(R_CONTROLX, LECTURA);
  Bloquear(&cola_dispX);
}
void interrupcionX() {
  *b++ = in(R_DATOSX);
  if (--tam_pet>0)
    out(R_CONTROLX, LECTURA);
  else
    Desbloquear(&cola_dispX);
}
```

```
// pto. entrada llamada al sistema escribeY
sis_escribeY(char *dir, int tam){
  char *buf = kmalloc(tam);
  copy_from_user_to_sys(dir, buf, tam);
  do_escribeY(buf, tam);
  kfree(buf); }
int tam_pet; char *b; tipo_cola cola_dispY;
do_escribeY(char *dirb, int tam){
  tam_pet = tam; b = dirb;
  out(R_DATOSY, *b++); out(R_CONTROLY, ESCR);
  Bloquear(&cola_dispY);}
void interrupcionY() {
  if (--tam_pet>0){
    out(R_DATOSY, *b++);out(R_CONTROLY, ESCR); }
  else Desbloquear(&cola_dispY);}
// pto. entrada llamada al sistema copiaXY
sis_copiaXY(int tam){
  char *buf = kmalloc(tam);
  do_leeX(buf, tam); do_escribeY(buf, tam);
  kfree(buf); }
```

a) Explique qué problema sucedería si se sustituye todo el código incluido en la función `sis_leeX` por la llamada `do_leeX(dir, tam)` identificando en qué sentencia del código se produciría este problema.

b) Explique qué problema sucedería si se sustituye todo el código incluido en la función `sis_escribeY` por la llamada `do_escribeY(dir, tam)` identificando en qué sentencia del código se produciría este problema.

c) Analicemos primero el modo de operación convencional: el proceso realiza una llamada al sistema para leer del dispositivo X y, justo después, invoca una llamada al sistema para escribir en el dispositivo Y. Detalle la traza de ejecución planteada a continuación identificando las activaciones del S.O. y los cambios de contexto suponiendo un núcleo no expulsivo.

- **P** (prioridad alta): solicita una lectura de 2 bytes del dispositivo X (`leeX`) especificando como *buffer* una variable global sin valor inicial que ocupa dos bytes almacenados al principio de una página que nunca se había accedido previamente. Completada la llamada de lectura, solicita una escritura en el dispositivo Y (`escribeY`) especificando esa misma variable. Finalmente, al retornar de la segunda llamada, **P** termina de forma normal.
- **Q** (prioridad baja): Recién creado, alternando fases de cálculo en modo usuario con llamadas `getpid`. Suponga que las sucesivas interrupciones encuentran alternativamente a este proceso en modo usuario y en sistema.

d) Estudiemos el modo de operación con copia directa: el proceso **P**, en vez invocar una llamada de lectura y, a continuación, una de escritura, realiza una única llamada para copiar dos bytes del dispositivo X al Y. Detalle la traza de ejecución de este nuevo escenario.

e) Una de las ventajas de la copia directa es reducir los accesos a memoria requeridos para llevar a cabo la operación. Para estimar la mejora, detalle qué accesos a memoria (operaciones `LOAD` y `STORE`) implica cada byte transferido en el modo de operación convencional y cuáles en el modo con copia directa.

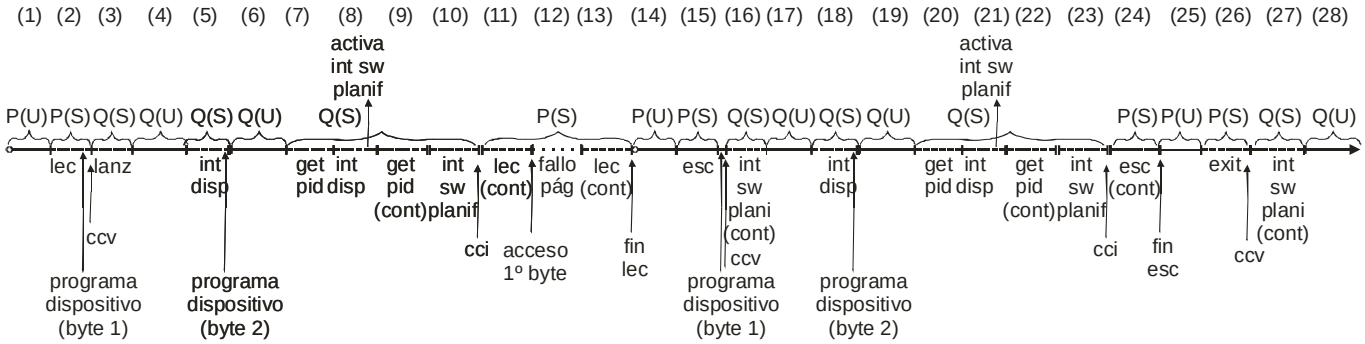
f) Una ventaja adicional de la copia directa es disminuir el número de activaciones del sistema operativo requeridas para completar la operación. Identifiqué qué tipo de activaciones del S.O. se reducirían al usar un modo de operación con copia directa frente a uno convencional: (i) nº de llamadas al sistema; (ii) nº de excepciones de fallo de página; (iii) nº de interrupciones de los dispositivos involucrados. ¿Se reduciría también (iv) el nº de cambios de contexto asociados a la programación de los dispositivos?

Solución

a) Si se realiza esa sustitución, en la variable global `b` quedará la dirección del *buffer* de usuario, de manera que cuando se produzca una interrupción del dispositivo `X` al completarse una operación de lectura de un byte, en la rutina de tratamiento de la misma se estará intentando acceder al mapa de usuario del proceso (`*b++ = in(R_DATOSX);`) para copiar el byte leído. Este acceso es erróneo puesto que desde el tratamiento de un evento asíncrono no se puede acceder al mapa de usuario del proceso puesto que es impredecible qué proceso está ejecutando en ese instante.

b) Si se realiza esa sustitución, en la variable global `b` quedará la dirección del *buffer* de usuario, de manera que cuando se produzca una interrupción del dispositivo `X` al completarse una operación de escritura de un byte, en la rutina de tratamiento de la misma se estará intentando acceder al mapa de usuario del proceso (`out(R_DATOSY, *b++);`) para enviar al dispositivo el siguiente byte a escribir. Este acceso es erróneo puesto que desde el tratamiento de un evento asíncrono no se puede acceder al mapa de usuario del proceso puesto que es impredecible qué proceso está ejecutando en ese instante.

c) La siguiente gráfica muestra la traza de ejecución del ejemplo planteado:

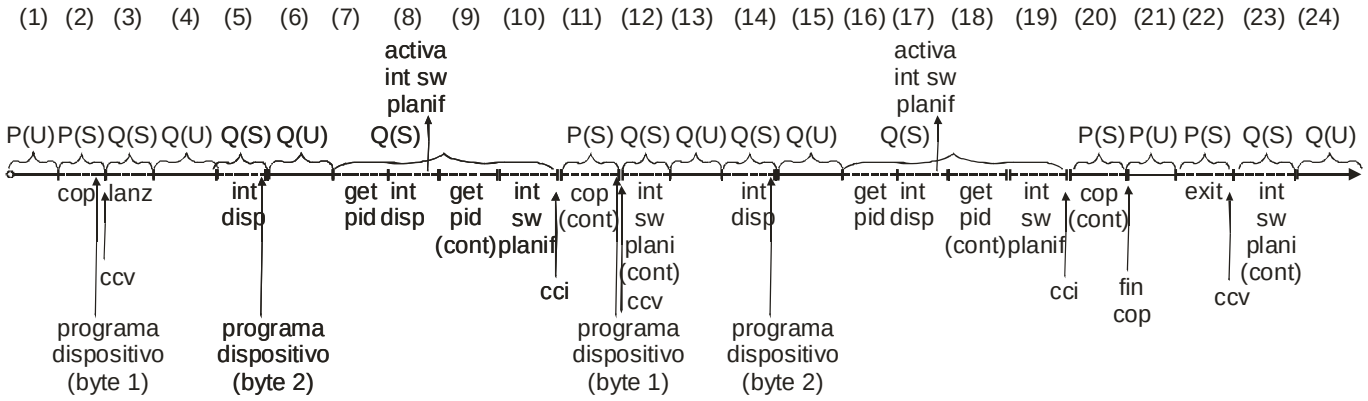


A continuación, se detallan cada uno de los eventos que ocurren durante la traza:

1. **P** en modo usuario realiza una llamada al sistema solicitando leer dos bytes del dispositivo X, especificando como *buffer* una variable global sin valor inicial que ocupa dos bytes y está almacenada en una única página no accedida previamente.
2. **P** en modo sistema programa el dispositivo para que lea el primer byte y se bloquea, 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. **Q** ejecuta en modo usuario hasta que se produce una interrupción del dispositivo X.
5. La rutina de interrupción del dispositivo copia el byte leído al buffer interno, detecta que quedan todavía datos por leer y programa la lectura del segundo byte en el dispositivo, completándose la rutina de interrupción.
6. **Q** continúa en modo usuario.
7. **Q** realiza la llamada al sistema `getpid`.
8. Mientras **Q** ejecuta en modo sistema la llamada `getpid`, se produce una interrupción del dispositivo X que, después de copiar el segundo byte al buffer interno, detecta que la operación se ha completado y desbloquea al proceso **P**. 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**.
9. Al tratarse de un núcleo no expulsivo, **Q** continúa y completa la llamada `getpid`.
10. 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.
11. **P** en modo sistema procede a copiar el primer byte almacenado en el *buffer* interno a la variable especificada por el programa en la llamada al sistema, pero, dado que la página que contiene esa variable no está residente, se produce un fallo de página que se anida con la llamada al sistema.
12. Al tratarse de una variable global sin valor inicial, estará incluida en una región sin soporte, por lo que no requerirá acceder a disco para traerla. El tratamiento del fallo de página buscará un marco libre, que suponemos que habrá, y simplemente lo rellenará con ceros, por motivos de seguridad, no requiriéndose ningún cambio de contexto.
13. **P** prosigue la llamada al sistema copiando el segundo byte, lo que no provocará fallo al estar la página ya residente, completando así la ejecución de la misma y retornando **P** a modo usuario.
14. **P** ejecuta en modo usuario hasta que realiza una segunda llamada al sistema para escribir en el dispositivo Y los dos bytes leídos.
15. **P** en modo sistema copia los bytes a escribir en un *buffer* interno (esta operación no debería provocar un fallo de página puesto que el *buffer* de usuario que contiene los dos bytes acaba de ser accedido), programa el dispositivo para que escriba el primer byte y se bloquea, produciéndose un cambio de contexto voluntario a **Q**.
16. **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.
17. **Q** ejecuta en modo usuario hasta que se produce una interrupción del dispositivo Y.
18. La rutina de interrupción del dispositivo detecta que quedan todavía datos por escribir, programa la escritura del segundo byte leído, completándose la rutina de interrupción.
19. **Q** continúa en modo usuario.
20. **Q** realiza la llamada al sistema `getpid`.

21. Mientras **Q** ejecuta en modo sistema la llamada `getpid`, se produce una interrupción del dispositivo Y que detecta que la operación se ha completado y desbloquea al proceso **P**. 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**.
22. Al tratarse de un núcleo no expulsivo, **Q** continúa y completa la llamada `getpid`.
23. 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.
24. **P** completa la llamada de escritura retornando a modo usuario.
25. **P** ejecuta en modo usuario hasta que realiza la llamada al sistema para completar su ejecución.
26. La llamada de terminación completa la ejecución de **P**, produciéndose un cambio de contexto voluntario a **Q**.
27. **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.
28. **Q** prosigue su ejecución en modo usuario.

d) La siguiente gráfica muestra 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 copiar dos bytes del dispositivo X al Y.
2. **P** en modo sistema programa el dispositivo para que lea el primer byte y se bloquea, 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. **Q** ejecuta en modo usuario hasta que se produce una interrupción del dispositivo X.
5. La rutina de interrupción del dispositivo copia el byte leído al buffer interno, detecta que quedan todavía datos por leer y programa la lectura del segundo byte en el dispositivo, completándose la rutina de interrupción.
6. **Q** continúa en modo usuario.
7. **Q** realiza la llamada al sistema `getpid`.
8. Mientras **Q** ejecuta en modo sistema la llamada `getpid`, se produce una interrupción del dispositivo X que, después de copiar el segundo byte al buffer interno, detecta que la operación se ha completado y desbloquea al proceso **P**. 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**.
9. Al tratarse de un núcleo no expulsivo, **Q** continúa y completa la llamada `getpid`.
10. 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.
11. **P** programa el dispositivo Y para que escriba el primer byte y se bloquea, produciéndose un cambio de contexto voluntario a **Q**.
12. **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.
13. **Q** ejecuta en modo usuario hasta que se produce una interrupción del dispositivo Y.
14. La rutina de interrupción del dispositivo detecta que quedan todavía datos por escribir, programa la escritura del segundo byte leído, completándose la rutina de interrupción.
15. **Q** continúa en modo usuario.
16. **Q** realiza la llamada al sistema `getpid`.
17. Mientras **Q** ejecuta en modo sistema la llamada `getpid`, se produce una interrupción del dispositivo Y que detecta que la operación se ha completado y desbloquea al proceso **P**. 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**.
18. Al tratarse de un núcleo no expulsivo, **Q** continúa y completa la llamada `getpid`.
19. 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.
20. **P** completa la llamada de escritura retornando a modo usuario.
21. **P** ejecuta en modo usuario hasta que realiza la llamada al sistema para completar su ejecución.
22. La llamada de terminación completa la ejecución de **P**, produciéndose un cambio de contexto voluntario a **Q**.

23. **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.
24. **Q** prosigue su ejecución en modo usuario.

e) En el modo de operación convencional, la lectura de un byte requiere un STORE para almacenarlo en el *buffer* interno (`*b++ = in(R_DATOSX);`) y, a continuación, un LOAD seguido de un STORE para copiarlo del *buffer* interno al *buffer* del usuario (`copy_from_sys_to_user(buf, dir, tam);`). En cuanto a la escritura, es justo lo complementario: un LOAD seguido de un STORE para copiarlo del *buffer* del usuario al *buffer* interno (`copy_from_user_to_sys(dir, buf, tam);`) y un LOAD para enviarlo al dispositivo (`out(R_DATOSY, *b++);`). En total, se producen tres operaciones LOAD y tres STORE por cada byte transferido.

En la modalidad de copia directa, se produce, en primer lugar, un STORE para almacenarlo en el *buffer* interno (`*b++ = in(R_DATOSX);`) y, a continuación, un LOAD para enviarlo al dispositivo (`out(R_DATOSY, *b++);`). En total, se producen una operación LOAD y una STORE por cada byte transferido.

f) Revisemos cada una de las opciones planteadas:

i. Llamadas al sistema: se reducen a la mitad puesto que en el método convencional hay que realizar una llamada para la lectura y otra para la escritura.
ii. Fallo de página: en el método convencional se pueden producir fallos de página en los accesos a los *buffers* de usuario, tanto al usado en la operación de lectura como en el de la escritura. Con la modalidad de copia directa, sólo se usan *buffers* internos que, en la mayoría de los sistemas operativos, siempre están residentes en memoria.
iii. Interrupciones de los dispositivos: se mantendrá el mismo número (sendas interrupciones por cada byte) puesto que la cantidad de datos a leer en el primer dispositivo y a escribir en el segundo es la misma con independencia de qué modalidad se use.
iv. Cambios de contexto asociados a la programación de dispositivos: será igual en ambos casos (dos bloqueos para toda la operación) por las mismas razones expuestas en el punto anterior.