Diseño de Sistemas Operativos. Febrero de 2012.
Ejercicio de procesos y memoria

Se pretende analizar el uso de *buffering* en las operaciones de escritura en un dispositivo. Para ello, se va a usar un hipotético dispositivo en el que las operaciones de escritura se hacen byte a byte y tal que cuando se completa la escritura de un byte, operación relativamente lenta, se genera una interrupción para notificarlo, quedando listo el dispositivo para iniciar una nueva operación. Considere que los siguientes dos fragmentos de pseudo-código corresponden a los del manejador de una operación de escritura sin *buffering* (`O_DIRECT`) y con *buffering*, respectivamente, sobre ese dispositivo en un sistema **monoprocesador** con un núcleo de tipo **no expulsivo**. Se trata de una versión simplificada que considera que el dispositivo lo usa de forma exclusiva un proceso (en la apertura se controlaría ese aspecto) y que obvia problemas de sincronización y control de errores.

Versión sin buffering

```
tipo_cola_procesos cola_disp;
int escritura_sin_buff(char *dir, int tam) {
     while (tam--) {
          out(R_DATOS, *(dir++));
          out(R_CONTROL, ESCRITURA);
          Bloquear(&cola_disp);
}    }
void interrupcion() {//igual para ambas versiones
     Desbloquear(&cola_disp); }
```

Versión con buffering

```
char *buf; tipo_cola_procesos cola_disp;
int escritura_con_buff(char *dir, int tam) {
     buf=reservar_memoria_kernel(tam);
     copiar_de_usuario(buf,dir,tam) ;
     while (tam--) {
          out(R_DATOS, *(buf++));
          out(R_CONTROL, ESCRITURA);
          Bloquear(&cola_disp);
}    }
```

Para comparar las versiones vamos a utilizar el siguiente ejemplo de ejecución de procesos:

- **P** (prioridad alta): solicita una escritura de 2 bytes en el dispositivo especificando como datos a escribir una variable global con valor inicial que ocupa dos bytes tal que el primero está almacenado al final de una página residente mientras que el segundo al comienzo de una página no residente a la que se había accedido previamente pero sin modificarla. Justo después de retornar de la llamada al sistema, **P** termina de forma normal.
- **Q** (prioridad baja): Proceso recién creado, que alterna fases de cálculo en modo usuario con llamadas al sistema no bloqueantes (**`getpid`**).
- Todas las interrupciones que se produzcan durante la traza se encontrarán con el proceso en ejecución en modo usuario, excepto la última interrupción del dispositivo que se producirá estando el proceso en ejecución en modo sistema. Además, se supone que hay suficientes marcos libres en el sistema para la ejecución de los procesos.

a) Detalle la traza de ejecución planteada para el caso de la versión sin *buffering*, identificando las activaciones del S.O., los cambios de contexto voluntarios e involuntarios, así como qué tipos de fallos de página se producen (si requieren leer del disco y, en caso de que así sea, si es del dispositivo de paginación o del sistema de ficheros).
b) Repita la traza planteada para la versión con *buffering*.
c) ¿Qué versión es más eficiente?
d) Suponga que el dispositivo realiza operaciones de escritura por bloques usando DMA en vez trabajar byte a byte. ¿Cuál de las versiones sería más problemática?
Para reducir el número de cambios de contexto que se producen durante la operación de escritura, se plantea usar un esquema donde la rutina de interrupción se encarga de reprogramar el dispositivo, resultando en estas dos nuevas versiones alternativas.

Versión sin buffering

```
tipo_cola_procesos cola_disp;
int tam_datos; char *dir_datos;
int escritura_sin_buff(char *dir, int tam) {
     tam_datos = tam; dir_datos = dir;
     out(R_DATOS, *(dir_datos++));
     out(R_CONTROL, ESCRITURA);
     Bloquear(&cola_disp);  }
void interrupcion() {//igual para ambas versiones
     if (--tam_datos > 0 ) {
          out(R_DATOS, *(dir_datos++));
          out(R_CONTROL, ESCRITURA); }
     else Desbloquear(&cola_disp); }
```

Versión con buffering

```
char *buf;
tipo_cola_procesos cola_disp;
int tam_datos; char *dir_datos;
int escritura_con_buff(char *dir, int tam) {
     buf=reservar_memoria_kernel(tam);
     copiar_de_usuario(buf,dir,tam) ;
     tam_datos = tam; dir_datos = buf;
     out(R_DATOS, *(dir_datos++));
     out(R_CONTROL, ESCRITURA);
     Bloquear(&cola_disp);
}
```

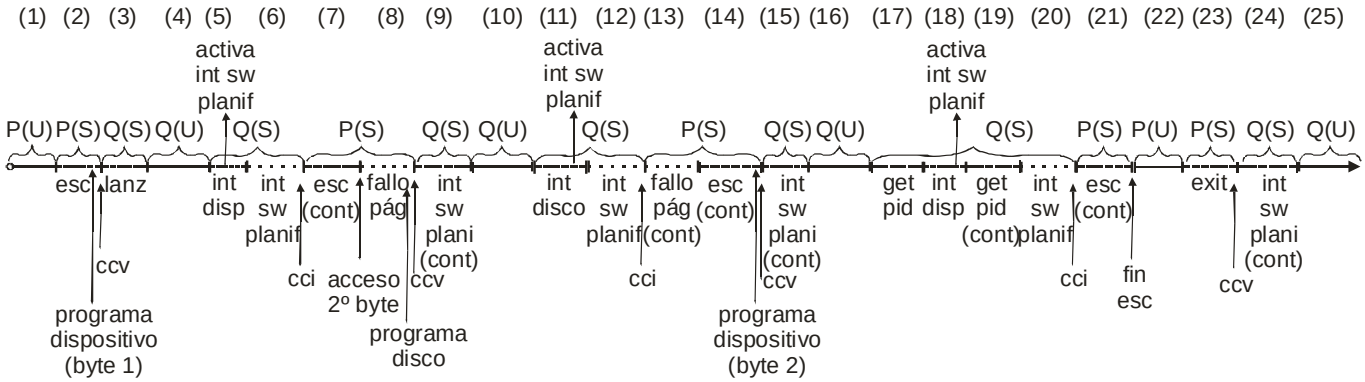
e) Identifique qué versión es errónea y explique por qué motivo.
f) Repita la traza planteada para la versión que no sea errónea.

Una ventaja del uso del *buffering* es que permite dotar de asincronismo a la operación de escritura retornando la llamada sin haberse completado la operación.

g) Explique qué cambios habría que hacer en el código del manejador para conseguir esta versión asíncrona.
h) Repita la traza planteada para esta versión asíncrona.

Solución

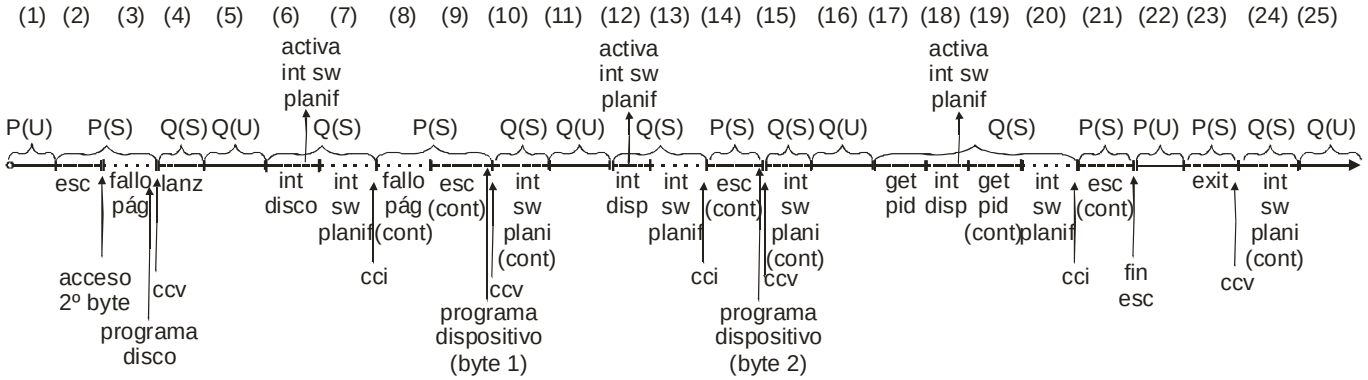
a) La siguiente gráfica muestra la traza de ejecución del ejemplo planteado:



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 escribir en el dispositivo, especificando como *buffer* una variable global con valor inicial que ocupa dos bytes y está almacenada en dos páginas: la primera residente y la segunda no residente habiéndose accedido previamente pero sin ser modificada.
2. **P** en modo sistema programa el dispositivo escribiendo el primer byte en el registro de datos del dispositivo, lo que no provoca un fallo de página puesto que la página que lo contiene está residente, 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. Durante la ejecución en modo usuario de **Q**, se produce una interrupción del dispositivo.
5. La rutina de interrupción del dispositivo 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**.
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 accede al segundo byte de la variable especificada por el programa en la llamada al sistema, pero, dado que la página que contiene ese byte no está residente, 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 en un fichero, y puesto que nunca ha sido modificada, habrá que leerla del fichero ejecutable que la contiene. Dado que hay marcos libres disponibles, no será necesario expulsar ninguna 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 se produce una interrupción del disco que indica el final de la lectura de la página.
11. El tratamiento de la interrupción del disco, entre otras operaciones, desbloquea el proceso **P** y, al ser éste más prioritario, activa una interrupción software de planificación expulsiva para forzar el cambio de contexto involuntario.
12. 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.
13. Se completa la rutina de fallo de página prosiguiendo la ejecución de la llamada.
14. **P** en modo sistema programa la escritura del segundo byte en el dispositivo y se bloquea, produciéndose un cambio de contexto voluntario a **Q**.
15. **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.
16. **Q** ejecuta en modo usuario hasta que realiza la llamada al sistema `getpid`.
17. Mientras **Q** ejecuta en modo sistema la llamada `getpid`, se produce una interrupción del dispositivo.
18. La rutina de interrupción del dispositivo 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**.
19. Al tratarse de un núcleo no expulsivo, **Q** continúa y completa la llamada `getpid`, activándose entonces el tratamiento de la interrupción software de planificación.
20. 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.
21. Se completa la llamada al sistema, retornando **P** a modo usuario.
22. **P** ejecuta en modo usuario hasta que realiza la llamada al sistema para completar su ejecución.
23. La llamada de terminación completa la ejecución de **P**, produciéndose un cambio de contexto voluntario a **Q**.
24. **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.
25. **Q** prosigue su ejecución en modo usuario.

b) La principal diferencia con el apartado anterior es que en este caso el fallo de página se produce al principio de la llamada, cuando se copian los datos a escribir en el *buffer* del núcleo que se ha reservado previamente. En cualquier caso, se vuelve a mostrar en la siguiente gráfica toda 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 escribir en el dispositivo, especificando como *buffer* una variable global con valor inicial que ocupa dos bytes y está almacenada en dos páginas: la primera residente y la segunda no residente habiéndose accedido previamente pero sin ser modificada.
2. **P** en modo sistema reserva un *buffer* de dos bytes dentro del mapa de memoria del núcleo y procede a realizar la copia. Al intentar acceder al segundo byte de la variable se produce un fallo de página que se anida con la llamada al sistema.
3. Al tratarse de una variable global con valor inicial, estará incluida en una región con soporte en un fichero, y puesto que nunca ha sido modificada, habrá que leerla del fichero ejecutable que la contiene. Dado que hay marcos libres disponibles, no será necesario expulsar ninguna página a memoria secundaria. En consecuencia, habrá que programar el disco para leer la página requerida, produciéndose un cambio de contexto voluntario a **Q**.
4. **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.
5. **Q** ejecuta en modo usuario hasta que se produce una interrupción del disco que indica el final de la lectura de la página.
6. El tratamiento de la interrupción del disco, entre otras operaciones, desbloquea el proceso **P** y, al ser éste más prioritario, activa una interrupción software de planificación expulsiva para forzar el cambio de contexto involuntario.
7. 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.
8. Se completa la rutina de fallo de página prosiguiendo la ejecución de la llamada.
9. **P** en modo sistema programa el dispositivo copiando el primer byte del *buffer* reservado al registro de datos del dispositivo, y se bloquea, produciéndose un cambio de contexto voluntario a **Q**.
10. **Q** reanuda su ejecución en el contexto de una rutina de tratamiento de una interrupción software de planificación donde quedó detenida su ejecución.
11. Durante la ejecución en modo usuario de **Q**, se produce una interrupción del dispositivo.
12. La rutina de interrupción del dispositivo 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**.
13. 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.
14. **P** en modo sistema programa la escritura del segundo byte en el dispositivo y se bloquea, produciéndose un cambio de contexto voluntario a **Q**.
15. **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.
16. **Q** ejecuta en modo usuario hasta que realiza la llamada al sistema `getpid`.
17. Mientras **Q** ejecuta en modo sistema la llamada `getpid`, se produce una interrupción del dispositivo.
18. La rutina de interrupción del dispositivo 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**.
19. Al tratarse de un núcleo no expulsivo, **Q** continúa y completa la llamada `getpid`, activándose entonces el tratamiento de la interrupción software de planificación.
20. 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.
21. Se completa la llamada al sistema, retornando **P** a modo usuario.
22. **P** ejecuta en modo usuario hasta que realiza la llamada al sistema para completar su ejecución.
23. La llamada de terminación completa la ejecución de **P**, produciéndose un cambio de contexto voluntario a **Q**.
24. **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.
25. **Q** prosigue su ejecución en modo usuario.

c) Comparando las trazas, se puede observar que se han producido los mismos eventos aunque en diferente orden. Sin embargo, el uso del *buffering* conlleva la reserva del *buffer* y la copia al mismo de los datos a escribir, por lo que la operación será menos eficiente.
Por otro lado, al tratarse de una operación de escritura relativamente lenta, como plantea el enunciado, en la versión sin *buffering*, cuanto mayor sea el tamaño de los datos a escribir, mayor será la probabilidad de que durante la ejecución de la llamada al sistema sea expulsada una página que contiene datos, lo que causaría fallos de página adicionales. Evidentemente, esto no ocurriría en la versión con *buffering*.
A la vista de la sobrecarga que conlleva el uso del *buffering*, puede parecer sorprendente que los sistemas operativos ofrezcan normalmente por defecto operaciones de entrada/salida con *buffering*, siendo necesario solicitar explícitamente que no se use esta técnica para una determinada operación si así se desea (con el valor `O_DIRECT` en los sistemas UNIX). En los siguientes apartados se analizarán algunas ventajas del uso de este mecanismo (como facilitar la utilización de DMA o la implementación de operaciones asíncronas), pero en este punto conviene resaltar una adicional que no aparece en el ejemplo planteado al considerar sólo una operación en cada momento: la técnica del *buffering* permite agrupar distintas operaciones, lo que permite optimizar de forma drástica el rendimiento de algunos dispositivos (como, por ejemplo, el disco o la red).

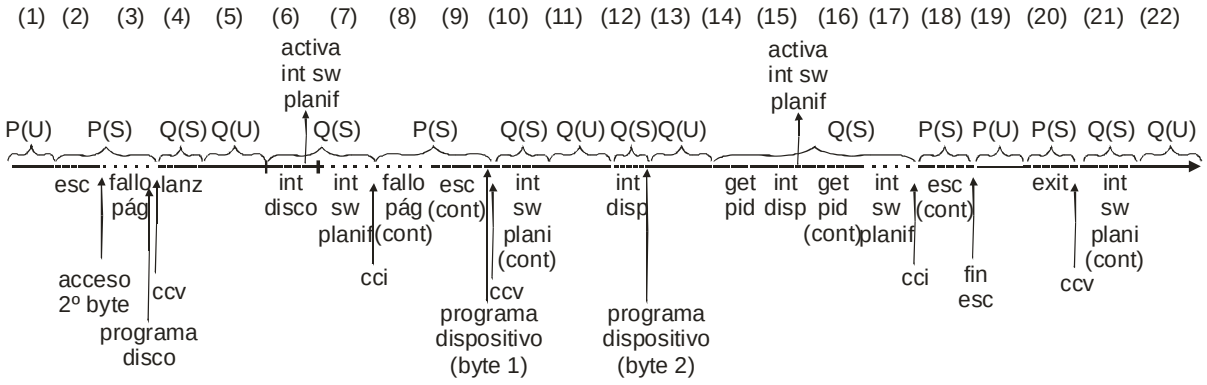
d) Cuando se realiza una operación de DMA, hay que asegurarse de que la zona de memoria involucrada en la misma permanece residente mientras se lleva a cabo la operación. Cuando se utiliza *buffering*, este requisito se satisface automáticamente puesto que la zona afectada por el DMA corresponde a memoria del mapa del sistema, que siempre está residente. Cuando no se usa *buffering*, sin embargo, el manejador debe marcar de forma explícita como "no expulsables" las páginas involucradas en el DMA, y, evidentemente, debe restaurar su estado previo al concluir la operación.

e) En el código de la versión sin *buffering* 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 leer el siguiente carácter que se pretende escribir):

```
if (--tam_datos > 0 ) { out(R_DATOS, *(dir_datos++)); ...
```

Obsérvese que en la versión sin *buffering* la variable `dir_datos` hace referencia al mapa de usuario, pero esto no es así en la versión con *buffering*, donde esa variable apunta al *buffer* reservado en el mapa de memoria del sistema al principio de la operación y, por lo tanto, no puede causar problemas.
Volviendo al problema identificado, hay que recordar que el carácter asíncrono de las interrupciones hace que sea imprevisible qué proceso está ejecutando cuando se realiza su tratamiento, por lo que el byte se leerá erróneamente del mapa de un proceso impredecible, no siendo posible, además, que esté en ejecución el proceso que quería escribir dicho carácter puesto que está bloqueado esperando que se complete la operación.

f) La traza de ejecución de la única versión correcta, la que usa *buffering*, es similar a la planteada en el segundo apartado, con la única diferencia de que en este caso se vuelve a programar el dispositivo en el contexto de la rutina de interrupción, lo que reduce el número de cambios de contexto requeridos para completar la operación. En cualquier caso, se vuelve a mostrar en la siguiente gráfica toda 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 escribir en el dispositivo, especificando como *buffer* una variable global con valor inicial que ocupa dos bytes y está almacenada en dos páginas: la primera residente y la segunda no residente habiéndose accedido previamente pero sin ser modificada.
2. **P** en modo sistema reserva un *buffer* de dos bytes dentro del mapa de memoria del núcleo y procede a realizar la copia. Al intentar acceder al segundo byte de la variable se produce un fallo de página que se anida con la llamada al sistema.
3. Al tratarse de una variable global con valor inicial, estará incluida en una región con soporte en un fichero, y puesto que nunca ha sido modificada, habrá que leerla del fichero ejecutable que la contiene. Dado que hay marcos libres disponibles, no será necesario expulsar ninguna página a memoria secundaria. En consecuencia, habrá que programar el disco para leer la página requerida, produciéndose un cambio de contexto voluntario a **Q**.
4. **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.
5. **Q** ejecuta en modo usuario hasta que se produce una interrupción del disco que indica el final de la lectura de la página.

6. El tratamiento de la interrupción del disco, entre otras operaciones, desbloquea el proceso **P** y, al ser éste más prioritario, activa una interrupción software de planificación expulsiva para forzar el cambio de contexto involuntario.
7. 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.
8. Se completa la rutina de fallo de página prosiguiendo la ejecución de la llamada.
9. **P** en modo sistema programa el dispositivo copiando el primer byte del *buffer* reservado al registro de datos del dispositivo, y se bloquea, produciéndose un cambio de contexto voluntario a **Q**.
10. **Q** reanuda su ejecución en el contexto de una rutina de tratamiento de una interrupción software de planificación donde quedó detenida su ejecución.
11. Durante la ejecución en modo usuario de **Q**, se produce una interrupción del dispositivo.
12. La rutina de interrupción del dispositivo detecta que quedan todavía datos por transferir y programa la escritura del segundo byte en el dispositivo, completándose la rutina de interrupción y retornando **Q** a modo usuario.
13. **Q** ejecuta en modo usuario hasta que realiza la llamada al sistema `getpid`.
14. Mientras **Q** ejecuta en modo sistema la llamada `getpid`, se produce una interrupción del dispositivo.
15. La rutina de interrupción del dispositivo detecta que no quedan más caracteres por transmitir 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**.
16. Al tratarse de un núcleo no expulsivo, **Q** continúa y completa la llamada `getpid`, activándose entonces el tratamiento de la interrupción software de planificación.
17. 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.
18. Se completa la llamada al sistema, retornando **P** a modo usuario.
19. **P** ejecuta en modo usuario hasta que realiza la llamada al sistema para completar su ejecución.
20. La llamada de terminación completa la ejecución de **P**, produciéndose un cambio de contexto voluntario a **Q**.
21. **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.
22. **Q** prosigue su ejecución en modo usuario.

g) En principio, para dotar de asincronismo a la operación de escritura en la última versión con *buffering* propuesta, bastaría con eliminar la operación de bloqueo dentro de la llamada y la operación de desbloqueo correspondiente dentro de la rutina de interrupción.

Versión con buffering

```
char *buf;
tipo_cola_procesos cola_disp;
int tam_datos; char *dir_datos;
int escritura_con_buff(char *dir, int tam) {
     buf=reservar_memoria_kernel(tam);
     copiar_de_usuario(buf,dir,tam) ;
     tam_datos = tam; dir_datos = buf;
     out(R_DATOS, *(dir_datos++));
     out(R_CONTROL, ESCRITURA);  }
```

```
void interrupcion() {
     if (--tam_datos > 0 ) {
          out(R_DATOS, *(dir_datos++));
          out(R_CONTROL, ESCRITURA);
     }
}
```

Sería conveniente, sin embargo, hacer algunas matizaciones, más allá de lo que es el ejercicio planteado. Por simplicidad, el enunciado supone que hay una política de control en la operación de apertura del dispositivo para asegurar que el dispositivo se usa de forma exclusiva por parte de un único proceso. Esta restricción simplifica el código al no tener que tratar la posibilidad de que llegue una nueva petición de escritura mientras se está procesando una operación previa. Sin embargo, al hacer en esta nueva versión que la operación de escritura devuelva el control al proceso sin haberse completado, aparecerá esta situación incluso aunque el dispositivo se use de forma exclusiva. La siguiente versión soluciona este problema usando una cola de operaciones pendientes de procesar:

Versión con buffering

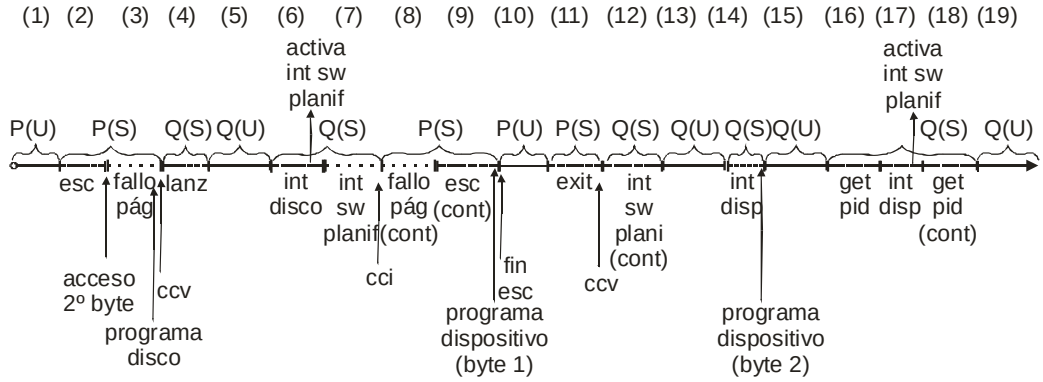
```
char *buf;
tipo_cola_procesos cola_disp;
struct op_t {int t_dat; char *d_dat; };
tipo_cola_ops cola_ops;
int escritura_con_buff(char *dir, int tam) {
     struct op_t op;
     buf=reservar_memoria_kernel(tam);
     copiar_de_usuario(buf,dir,tam);
     op.t_dat = tam; op.d_dat = buf;
     encolar_op(op, &cola_ops);
     // Comprueba si cola previamente vacía
     if (cola_ops.primera==cola_ops.ultima) {
       out(R_DATOS, *(op.d_dat++));
       out(R_CONTROL, ESCRITURA);
     }
}
```

```
void interrupcion() {
     // ¿fin de operación en curso?
     if (cola_ops.primera->t_dat==0)
       desencolar_primera_op(&cola_ops);

     // Sólo no reprograma si fin op. actual
     // y no hay más pendientes
     if ((cola_ops.primera) &&
      (--cola_ops.primera->t_dat > 0)) {
       out(R_DATOS,*(cola_ops.primera->d_dat++));
       out(R_CONTROL, ESCRITURA);
     }
}
```

Esta nueva versión no sólo resuelve el problema planteado por las múltiples peticiones que puede realizar un único proceso, sino que se podría aplicar a un dispositivo que no se utilice en modo exclusivo al que llegaran peticiones de distintos procesos. En el esquema usado cada petición simplemente se encola y sólo programa el dispositivo si éste estaba "parado". Por su parte, la rutina de interrupción, mientras haya trabajo pendiente, se encarga de reprogramar la siguiente transferencia de la operación actual o, si ésta se ha completado, la primera de la siguiente.

h) La principal diferencia de esta traza de ejecución con la previamente especificada es que el proceso que solicita la escritura no se bloquea e incluso termina su ejecución sin haberse completado la escritura en el dispositivo. 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 escribir en el dispositivo, especificando como *buffer* una variable global con valor inicial que ocupa dos bytes y está almacenada en dos páginas: la primera residente y la segunda no residente habiéndose accedido previamente pero sin ser modificada.
2. **P** en modo sistema reserva un *buffer* de dos bytes dentro del mapa de memoria del núcleo y procede a realizar la copia. Al intentar acceder al segundo byte de la variable se produce un fallo de página que se anida con la llamada al sistema.
3. Al tratarse de una variable global con valor inicial, estará incluida en una región con soporte en un fichero, y puesto que nunca ha sido modificada, habrá que leerla del fichero ejecutable que la contiene. Dado que hay marcos libres disponibles, no será necesario expulsar ninguna página a memoria secundaria. En consecuencia, habrá que programar el disco para leer la página requerida, produciéndose un cambio de contexto voluntario a **Q**.
4. **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.
5. **Q** ejecuta en modo usuario hasta que se produce una interrupción del disco que indica el final de la lectura de la página.
6. El tratamiento de la interrupción del disco, entre otras operaciones, desbloquea el proceso **P** y, al ser éste más prioritario, activa una interrupción software de planificación expulsiva para forzar el cambio de contexto involuntario.
7. 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.
8. Se completa la rutina de fallo de página prosiguiendo la ejecución de la llamada.
9. **P** en modo sistema programa el dispositivo copiando el primer byte del *buffer* reservado al registro de datos del dispositivo, completando la llamada y retornando a modo usuario.
10. **P** ejecuta en modo usuario hasta que realiza la llamada al sistema para completar su ejecución.
11. La llamada de terminación completa la ejecución de **P**, 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. Durante la ejecución en modo usuario de **Q**, se produce una interrupción del dispositivo.
14. La rutina de interrupción del dispositivo detecta que quedan todavía datos por transferir y programa la escritura del segundo byte en el dispositivo, completándose la rutina de interrupción y retornando **Q** a modo usuario.
15. **Q** ejecuta en modo usuario hasta que realiza la llamada al sistema `getpid`.
16. Mientras **Q** ejecuta en modo sistema la llamada `getpid`, se produce una interrupción del dispositivo.
17. La rutina de interrupción del dispositivo detecta que no quedan más caracteres por transmitir, completándose su ejecución.
18. **Q** completa la llamada `getpid`, retornando a modo usuario.
19. **Q** prosigue su ejecución en modo usuario.