Diseño de Sistemas Operativos. Febrero de 2013.
Ejercicio de procesos y memoria.

Se pretende analizar la problemática específica que presenta la programación de dispositivos que usan DMA. Para ello, se plantea el caso de una operación de **lectura** de un hipotético dispositivo (p. ej., una unidad de cinta) que utiliza DMA con un tamaño máximo de unidad de transferencia (*TU*) de 512 bytes por cada operación. Considere un sistema operativo con un tamaño de página (*TP*) de 4KiB y que mantiene el mapa de memoria del sistema siempre residente. Suponga que el siguiente fragmento de pseudo-código corresponde al manejador de la operación de lectura sobre ese dispositivo en un sistema **monoprocesador** con un núcleo de tipo **no expulsivo**. Esta función de lectura ofrece dos modos de operación: con *buffering* (DMA del dispositivo a un *buffer* del sistema), que es la opción por defecto de la mayoría de los SS.OO., y sin *buffering* (DMA del dispositivo **directamente** al *buffer* del usuario: `O_DIRECT`). Se trata de una versión simplificada que considera que el dispositivo lo usa de forma exclusiva un proceso y que obvia problemas de sincronización y control de errores.

```
tipo_cola_procesos cola_disp;
int lectura(void *dir, int tam, int modo) {
     void *d, *direc; int t;
     direc=dir;
     if (modo == O_DIRECT) {
          touch_pages(dir, tam); // más adelante
          lock_pages(dir, tam); // no mostrada  }
     else
          d = reservar_memoria_kernel(TU);
     while (tam>0) {
          if (modo == O_DIRECT) d = direc;
          t = min(tam, TU);
          programar(d, t); // más adelante
          Bloquear(&cola_disp);
          if (modo != O_DIRECT)
               copiar_a_usuario(direc,d,t);
          tam-=t; direc+=t;} // fin while
          if (modo == O_DIRECT)
               unlock_pages(dir, tam); // no mostrada
          else
            liberar_memoria_kernel(d);}// fin lectura
     void programar(void *di, int ta) {
          out(R_DMA_DIR, physical(di));// no mostrada
          out(R_DMA_TAM, ta);
          out(R_DMA_CTRL, LECTURA);}
     void touch_pages(void *di, int ta) {
          int np= (di+ta)/TP – (di/TP) + 1;
          for ( ; np>0; np--, di+= TP)
               *di; // accede a esa posición
     }
     void interrupcion() {
          Desbloquear(&cola_disp); }
```

Para comparar los dos modos de operación vamos a utilizar el siguiente ejemplo de ejecución de procesos:

- **P** (prioridad alta): solicita una lectura de 1000 bytes del dispositivo especificando como datos a leer una variable global sin valor inicial tal que sus primeros 600 bytes están almacenados al final de una página residente mientras que los 400 restantes al comienzo de una página no residente a la no que se había accedido previamente. Justo después de retornar de la llamada al sistema, **P** termina de forma normal. Suponga que hay marcos libres suficientes.
- **Q** (prioridad baja): Proceso recién creado, que alterna cálculos en modo usuario con llamadas 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.

a) Detalle la traza de ejecución planteada para el caso del modo con *buffering* (`!=O_DIRECT`), identificando las activaciones del S.O., los cambios de contexto voluntarios e involuntarios, así como los fallos de página que se producen.

b) Explique qué labor deberían llevar a cabo las funciones *touch_pages*, *lock_pages*, *unlock_pages* y *physical*. ¿Por qué las tres primeras sólo se usan para el caso de E/S directa y la cuarta en ambos casos?

c) Repita la traza planteada para la versión sin *buffering* (`==O_DIRECT`).

Para reducir el número de cambios de contexto que se producen durante la operación de lectura, se plantea usar un esquema donde la rutina de interrupción se encarga de reprogramar el dispositivo, resultando en esta nueva versión:

```
tipo_cola_procesos cola_disp;
void *d, *direc; int t;
int lectura(void *dir, int tam, int modo) {
     direc=dir;
     if (modo == O_DIRECT) {
          touch_pages(dir,tam);
          lock_pages(dir,tam); d = direc;}
     else d=reservar_memoria_kernel(TU); //fin if
     t = min(tam, TU);  programar(d, t);
     Bloquear(&cola_disp);
     if (modo == O_DIRECT)
          unlock_pages(dir, tam);
     else liberar_memoria_kernel(d);   }
     void programar(void *di, int ta) {//igual }
     void touch_pages(void *di, int ta) {//igual}
     void interrupcion() {
          if (modo != O_DIRECT)
               copiar_a_usuario(direc,d,t);
          tam-=t; direc+=t;
          if (tam>0) {
               if (modo == O_DIRECT) d = direc;
               t = min(tam, TU);
               programar(d, t); }
          else Desbloquear(&cola_disp); }
```

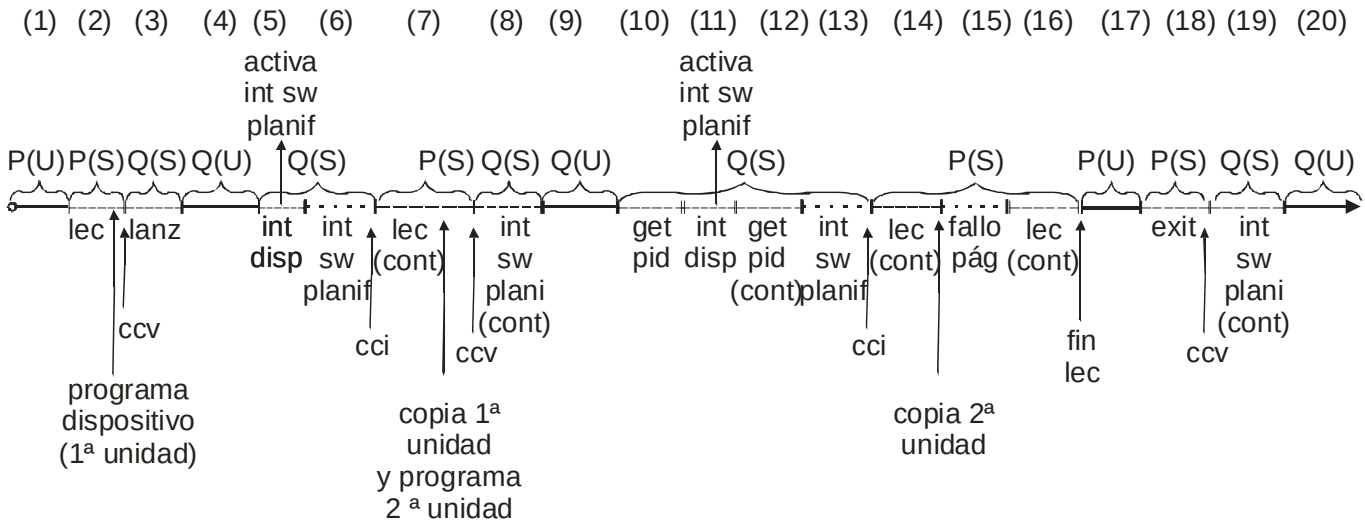
d) ¿Es correcto el funcionamiento del modo de operación con *buffering*? En caso negativo, explique por qué motivo y describa brevemente una posible solución.

e) La misma cuestión para el modo de operación directo.

f) Algunas arquitecturas, como MIPS o SPARC, no aseguran la coherencia de la cache de memoria cuando se realizan operaciones de DMA (es decir, los datos viajan desde el dispositivo a memoria o viceversa sin afectar a la cache). ¿Qué problemas podrían ocurrir en estos sistemas durante la operación de lectura planteada en la primera sección? Suponiendo que existe una operación para volcar de la cache a memoria los datos modificados presentes en la misma correspondiente a un rango de direcciones (*volcar_cache(dir, tam)*) y otra para invalidarlos (*invalidar_cache(dir, tam)*), explique cómo las usaría mostrando en qué parte del código se incluirían.

Solución

a) La siguiente gráfica muestra la traza de ejecución del ejemplo planteado para el caso con *buffering*:



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

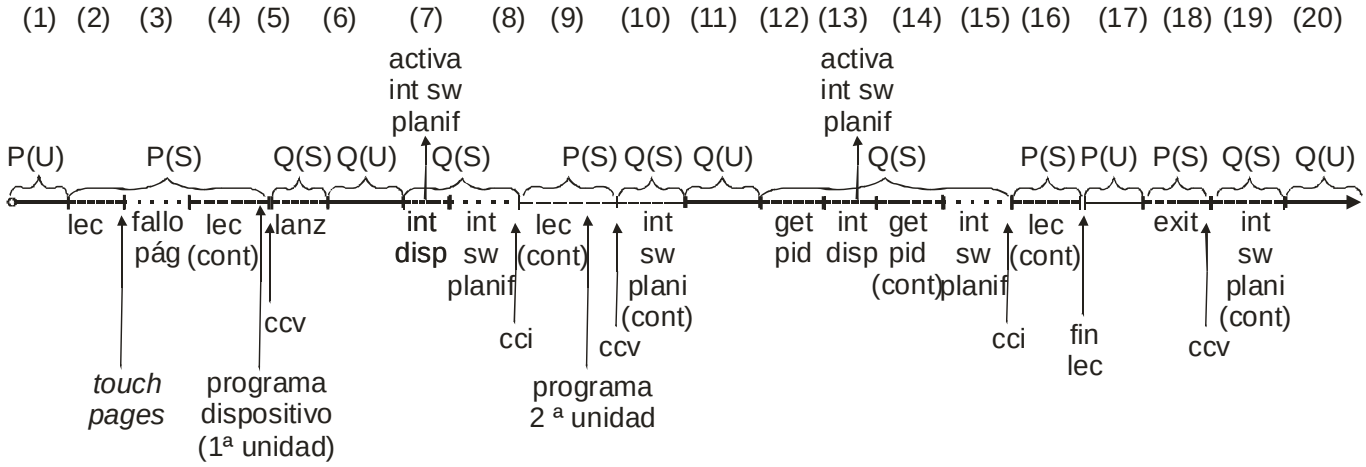
1. **P** en modo usuario realiza una llamada al sistema solicitando leer del dispositivo, especificando como *buffer* una variable global sin valor inicial tal que sus primeros 600 bytes están almacenados al final de una página residente mientras que los 400 restantes al comienzo de una página no residente a la no que se había accedido previamente.
2. **P** en modo sistema reserva un *buffer* del sistema, programa la operación de DMA del dispositivo especificando la dirección del *buffer* del sistema y el tamaño de los datos a transferir (512 bytes) 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 realiza la copia los datos leídos del *buffer* del sistema al especificado por el programa, no produciéndose fallo de página al estar esa primera página residente. A continuación, programa la segunda transferencia (488 bytes) y se bloquea, produciéndose un cambio de contexto voluntario a **Q**.
8. **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.
9. **Q** ejecuta en modo usuario hasta que realiza la llamada al sistema `getpid`.
10. Mientras **Q** ejecuta en modo sistema la llamada `getpid`, se produce una interrupción del dispositivo.
11. 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**.
12. Al tratarse de un núcleo no expulsivo, **Q** continúa y completa la llamada `getpid`, activándose entonces el tratamiento de la interrupción software de planificación.
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 procede a copiar los nuevos datos leídos del *buffer* del sistema a la parte correspondiente del especificado por el programa, produciéndose un fallo de página al alcanzarse el principio de la segunda página asociada al *buffer* de usuario.
15. 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 habrá según plantea el enunciado, y simplemente lo rellenará con ceros, por motivos de seguridad, no requiriéndose ningún cambio de contexto.
16. Se completa la llamada al sistema, retornando **P** a modo usuario.
17. **P** ejecuta en modo usuario hasta que realiza la llamada al sistema para completar su ejecución.
18. La llamada de terminación completa la ejecución de **P**, produciéndose un cambio de contexto voluntario a **Q**.
19. **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.
20. **Q** prosigue su ejecución en modo usuario.

b) Para entender qué labor debería llevar a cabo cada una de estas funciones, hay que tener en cuenta que la técnica del DMA trabaja usando direcciones físicas. Eso implica, por un lado, que hay que especificar una dirección física a la hora de programar una operación por DMA. Por otro lado, ese modo de funcionamiento requiere que la zona de memoria involucrada en una operación de DMA no se vea afectada por el esquema de memoria virtual, es decir, que hay que garantizar que las páginas implicadas están residentes en memoria durante toda la operación de DMA. Si la zona involucrada por una operación de DMA pertenece al mapa de memoria del sistema, este requisito de "residencia" se cumple automáticamente. Sin embargo, si se trata de una zona incluida en el mapa de memoria de usuario, hay que asegurarse explícitamente de que las páginas están residentes durante toda la operación.
Hecha esta aclaración, a continuación, se explica la labor realizada por cada una de esas cuatro funciones

- *touch_pages*: se encarga de acceder a cada una de las páginas del *buffer* de usuario para asegurarse de que estén residentes en memoria, provocando fallos de página en caso de que no lo estuvieran previamente.
- *lock_pages*: establece que las páginas afectadas no pueden ser reemplazadas por el sistema de gestión de memoria ("retiene" las páginas en memoria).
- *unlock_pages*: realiza la acción contraria, es decir, anotar que esas páginas ya pueden ser reemplazadas.
- *physical*: obtiene la dirección física asociada a una dirección lógica del proceso actualmente en ejecución que corresponde a una página residente (es decir, retorna a qué dirección física hace referencia la entrada de la tabla de páginas asociada a esa dirección lógica).

Tal como se explicó al inicio de este apartado, las tres primeras funciones sólo se requieren para el caso de entrada/salida directa (sin *buffering*), donde es necesario asegurar la residencia de las páginas durante la operación de DMA.
En cuanto a la función que obtiene la dirección física, es necesaria en cualquier caso puesto que tanto las direcciones lógicas de usuario como las del sistema no se corresponden directamente con direcciones físicas (recuerde que en un sistema con MMU, dentro del procesador nunca se manejan direcciones físicas).

c) La siguiente gráfica muestra la traza de ejecución del ejemplo planteado para el caso sin *buffering*:



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 del dispositivo, especificando como *buffer* una variable global sin valor inicial tal que sus primeros 600 bytes están almacenados al final de una página residente mientras que los 400 restantes al comienzo de una página no residente a la no que se había accedido previamente.
2. **P** en modo sistema accede a cada una de las páginas del *buffer* de usuario usando la función *touch_pages*. Al acceder a la segunda página, se produce un fallo de página.
3. 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 habrá según plantea el enunciado, y simplemente lo rellenará con ceros, por motivos de seguridad, no requiriéndose ningún cambio de contexto.
4. **P** continúa la llamada programando la operación de DMA del dispositivo especificando la dirección del *buffer* del usuario y el tamaño de los datos a transferir y se bloquea, produciéndose un cambio de contexto voluntario a **Q**.
5. **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.
6. Durante la ejecución en modo usuario de **Q**, se produce una interrupción del dispositivo.
7. 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**.
8. 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.
9. **P** en modo sistema programa la segunda transferencia 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. Al completarla, retorna a modo usuario.
11. **Q** ejecuta en modo usuario hasta que realiza la llamada al sistema `getpid`.
12. Mientras **Q** ejecuta en modo sistema la llamada `getpid`, se produce una interrupción del dispositivo.
13. 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**.
14. Al tratarse de un núcleo no expulsivo, **Q** continúa y completa la llamada `getpid`, activándose entonces el tratamiento de la interrupción software de planificación.
15. 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.
16. Se completa la llamada al sistema, retornando **P** a modo usuario.
17. **P** ejecuta en modo usuario hasta que realiza la llamada al sistema para completar su ejecución.
18. La llamada de terminación completa la ejecución de **P**, produciéndose un cambio de contexto voluntario a **Q**.
19. **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.
20. **Q** prosigue su ejecución en modo usuario.

Hay que aclarar que, por simplicidad, se ha supuesto que las dos páginas donde está almacenado el *buffer* de usuario están contiguas en memoria física, aunque no sea realista. Hay que tener en cuenta que, dado que la técnica de DMA usa direcciones físicas, no se puede realizar una transferencia en una única operación si la zona afectada está almacenada en dos o más páginas no contiguas. Así, para el caso de entrada/salida directa, en el cálculo del tamaño de la unidad de transferencia de cada operación (en cada iteración del bucle), si no se puede garantizar que las páginas implicadas estén contiguas, habría que asegurarse de que la unidad de transferencia no atraviesa la frontera entre dos páginas:

```
resto_pag = TP – (direc % TP);
t = min(tam, TU, resto_pag);
```

Como consecuencia de este cambio, la operación planteada en la traza requeriría tres transferencias (de tamaños 512, 88 y 400 bytes) en vez de dos.

Nótese que en la versión con *buffering* se ha asumido que el espacio contiguo en direcciones lógicas reservado por la función `reservar_memoria_kernel` es además contiguo en direcciones físicas (es decir, corresponde en Linux a la función `kmalloc` y no a `vmalloc`) puesto que en caso contrario se presentaría este mismo problema.

d) En el código de la versión con *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 copiar los datos leídos por DMA desde el *buffer* del sistema al del usuario (`copiar_a_usuario`).
Una solución es reservar inicialmente en la función de lectura un *buffer* del sistema con el tamaño suficiente para almacenar todos los datos que se pretenden leer, en vez de sólo con el tamaño de una unidad de transferencia, de manera que las transferencias por DMA programadas desde la rutina de interrupción vayan llenando sucesivamente ese *buffer* y que cuando se complete el mismo, se haga la copia entera del *buffer* de sistema al del usuario en el contexto de la función de lectura.

e) En la versión sin *buffering* no se produce el mismo problema que en la versión anterior puesto que no hay acceso al mapa de memoria de usuario en el contexto de la rutina de interrupción, ya que es el propio hardware de DMA el que se encarga de hacer la transferencia. Sin embargo, hay un aspecto problemático en el uso de la función que devuelve la dirección física asociada a una lógica (*physical*), ya que lo hará teniendo en cuenta el mapa de memoria (es decir, la tabla de páginas) del proceso actual y no el del proceso que solicitó la operación.
Una posible solución sería al inicio de la función de lectura guardar en una estructura de datos asociada a la petición las direcciones físicas correspondientes al *buffer* del usuario y usar estas direcciones físicas en la rutina de interrupción para programar la operación de DMA.
Una solución alternativa sería establecer al inicio de la operación de lectura, justo después de hacer que las páginas del *buffer* de usuario estén residentes mediante *touch_pages* y *lock_pages*, una asociación entre un rango de direcciones lógicas de sistema y los marcos de página que contienen el *buffer* del usuario. En la rutina de interrupción se usarían estas direcciones lógicas de sistema que son independientes de qué proceso esté ejecutando en cada momento.

f) En un sistema donde el hardware no garantiza la coherencia de la caché cuando se realizan operaciones de DMA; el software debe llevar a cabo acciones para asegurarla. En el caso de una operación de lectura por DMA, en un sistema sin garantía de coherencia, cuando un programa intente acceder a los datos leídos una vez que se ha completado la operación, puede obtener los valores que estaban previamente almacenados en la caché en vez de los leídos del dispositivo. La solución habitual es invalidar todas las entradas de la caché vinculadas con la zona de memoria afectada por la operación antes de intentar acceder a los datos leídos. Para resolver este problema en el fragmento de código planteado al inicio del enunciado, se puede incluir la invalidación justo al desbloquearse al completarse la operación:

```
Bloquear(&cola_disp);
Invalidar(d,t); ....
```