Diseño de Sistemas Operativos. Junio de 2009.
Ejercicio de procesos y memoria

Para conseguir una comunicación entre procesos eficiente, el sistema operativo intenta minimizar el número de operaciones de copia entre zonas de memoria que debe llevar a cabo para implementarla, buscando idealmente un esquema de *zero-copy* (este término corresponde a un sistema donde el sistema operativo/el procesador no realiza ninguna operación de copia de una zona de memoria a otra). En este ejercicio, se analizan algunos aspectos vinculados con esta problemática, centrándose en la comunicación entre procesos locales.

a) Explique razonadamente para qué tipo de arquitectura del sistema operativo es especialmente importante la optimización de esta operación. Cite al menos un ejemplo de sistema operativo que tenga este tipo de arquitectura.

b) Considere un mecanismo de comunicación entre procesos locales basado en el concepto de puerto y tal que el sistema operativo asigna un conjunto de *buffers* del sistema a cada puerto asociándolos a la estructura de datos que representa el estado de un puerto. El modo de operación de este mecanismo responde al siguiente esquema (muy simplificado):

```
enviar(puerto, mens, tam) {
   Busca buffer libre asociado a puerto;
   Copia mens a buffer;
   Si proceso esperando datos
      Desbloquea_proc(cola asociada a puerto);
}
```

```
recibir(puerto, mens, tam) {
   Si no hay datos asociados al puerto
      Bloquea_proc(cola asociada a puerto);
   Copia de buffer con datos a mens;
}
```

Para analizar cómo se comporta este esquema, desarrolle la traza de ejecución que se plantea a continuación, identificando las activaciones del sistema operativo, los cambios de contexto, distinguiendo entre voluntarios e involuntarios, así como qué tipos de fallos de página se producen. Se debe resaltar en la traza cuándo el sistema operativo realiza copias de memoria. Considere un núcleo no expulsivo, con *buffering* de páginas.

- **S** (prioridad alta; en ejecución): pide recibir de un puerto, que inicialmente no tiene datos, especificando como *buffer* de recepción una variable global sin valor inicial almacenada en una página a la que no se había accedido previamente.
- **C** (prioridad baja; listo para ejecutar expulsado por **S**): pide enviar al mismo puerto un mensaje de igual tamaño que el que pretende recibir **S**, especificando como *buffer* de envío una zona de memoria que está residente en ese momento.

c) Para intentar minimizar las copias, se plantea la siguiente versión alternativa:

```
enviar(puerto, mens, tam) {
   Si proceso esperando datos {
      Copia de mens a puerto->dir_destino;
      Desbloquea_proc(cola asociada a puerto);
   }
   Sino {
      Busca buffer libre asociado a puerto;
      Copia mens a buffer;
} }
```

```
recibir(puerto, mens, tam) {
   Si hay datos asociados al puerto
      Copia de buffer con datos a mens;
   Sino {
      puerto->dir_destino = mens;
      Bloquea_proc(cola asociada a puerto);
   }
}
```

Identifique, en primer lugar, qué situación intenta optimizar esta nueva versión. A continuación, explique por qué motivo esta solución no es válida indicando el error de diseño presente en la misma.

d) En esta última versión, en vez de asignarle un conjunto de *buffers* de sistema a cada puerto y almacenar en ellos los datos enviados al mismo, la estructura de datos que representa el estado de un puerto permite guardar información de qué páginas contienen los datos enviados al puerto.

```
enviar(puerto, mens, tam) {
   Trae a m. física páginas de mens no residentes;
   Marca páginas como no expulsables;
   Asocia marcos de página de mens con el puerto;
   Si proceso esperando datos
      Desbloquea_proc(cola asociada a puerto);
}
```

```
recibir(puerto, mens, tam) {
   Si no hay datos asociados al puerto
      Bloquea_proc(cola asociada a puerto);
   Copia de marcos asociados con el puerto a mens;
   Marca páginas copiadas como expulsables;
   Desasocia del puerto los marcos copiados;
}
```

d1) ¿Cómo cambiaría la traza del segundo apartado usando esta nueva versión? ¿Cuántas operaciones de copia de memoria se producirían en este caso?

d2) ¿Qué problema ocurriría si el proceso emisor, justo después de enviar un mensaje *M* pero antes de que otro proceso haya invocado *recibir*, modifica los datos contenidos en la variable *M*? ¿Qué técnica se puede usar para eliminarlo?

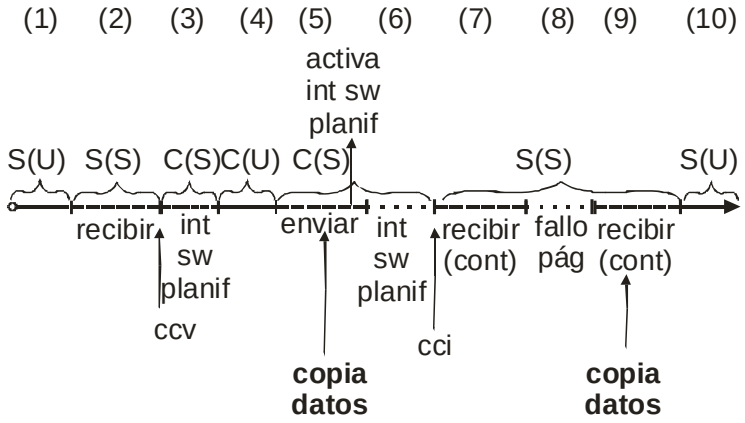
d3) ¿Qué restricciones debe cumplir el *buffer* que especifica el proceso emisor en la solicitud de envío?

d4) Describa cómo se podría implementar una versión *zero-copy* de este mecanismo.

Solución

a) Como es evidente, todo sistema operativo intenta optimizar las operaciones de comunicación entre procesos que residen en la misma máquina. Sin embargo, esta optimización tiene especial relevancia en el caso de los sistemas operativos con una arquitectura basada en un micro-núcleo. En este tipo de sistemas, la mayor parte de la funcionalidad del sistema operativo está implementada por servidores que ejecutan en modo usuario. Las aplicaciones, por su parte, acceden a la mayoría de los servicios del sistema enviando mensajes a los servidores correspondientes. Por tanto, en los sistemas operativos con una arquitectura basada en micro-núcleo, el grado de eficiencia de la comunicación entre procesos afecta a todas las aplicaciones del sistema, mientras que, en los sistemas monolíticos, sólo a aquéllas que usan explícitamente los servicios de comunicación entre procesos.
Uno de los sistemas basados en un micro-núcleo más relevante es Mach, desarrollado en Carnegie-Mellon a partir de 1985, tanto por su repercusión en la investigación sobre el diseño de sistemas operativos como por existir numerosos sistemas basados en el mismo (como OSF/1, Mac OS X y GNU Hurd). Es interesante resaltar que uno de los puntos críticos durante toda la evolución de Mach ha sido precisamente la mejora en la eficiencia del sistema de comunicación de procesos, que ha sido constantemente uno de los principales cuellos de botella de este sistema.

b) La siguiente figura muestra la traza de ejecución planteada en el enunciado:



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

1. **S** en modo usuario realiza la llamada al sistema `recibir` especificando como *buffer* una variable global sin valor inicial almacenada en una página a la que no se había accedido previamente.
2. **S** en modo sistema comprueba que no hay datos asociados al puerto especificado, bloqueándose en la cola de espera asociada al puerto y produciéndose un cambio de contexto voluntario a **C**.
3. **C** reanuda su ejecución en modo sistema en el contexto de la rutina de tratamiento de la interrupción software donde quedó detenida su ejecución al ser expulsado previamente por **S**. Al completarla, retorna a modo usuario, puesto que se trata de un núcleo no expulsivo y no es posible que se hubiera quedado en medio de una llamada al sistema.
4. **C** en modo usuario realiza la llamada al sistema `enviar` especificando como *buffer* una zona de memoria residente. Nótese que esta suposición es bastante razonable puesto que, normalmente, un proceso emisor rellena de datos el *buffer* que pretende enviar justo antes de invocar la llamada al sistema de envío, por lo que es bastante probable que la zona de memoria del *buffer* esté residente en memoria.
5. **C** en modo sistema encuentra un *buffer* asociado al puerto que está libre copiando en el mismo los datos que se pretenden enviar. En esta primera **copia de memoria** no se produce ningún fallo de página al estar el *buffer* de usuario residente. Al detectar que hay un proceso esperando por datos en ese puerto, lo desbloquea. Dado que el proceso desbloqueado es más prioritario que el actual, se activa una interrupción software de planificación para llevar a cabo el cambio de contexto involuntario de **C** a **S**. Como el núcleo es no expulsivo, **C** completará la llamada antes de dispararse la interrupción software.
6. La rutina de interrupción software de planificación lleva a cabo el cambio de contexto involuntario requerido, con lo que **S** continúa la ejecución de la llamada al sistema justo en el punto donde quedó detenida.
7. **S** en modo sistema comienza la copia de datos desde el *buffer* del puerto al del usuario especificado en la llamada, pero, dado que no se había accedido previamente a la página que contiene el *buffer*, se produce un fallo de página que se anida con la llamada al sistema.
8. Al tratarse de una variable global sin valor inicial, estará incluida en una página de una región anónima, sin soporte, por lo que no será necesario leerla desde memoria secundaria. Además, al tratarse de un sistema con *buffering* de páginas, habrá marcos libres disponibles. En consecuencia, no habrá necesidad de acceder al disco para servir ese fallo de página, no requiriendo, por tanto, ningún cambio de contexto.
9. Al finalizar el tratamiento del fallo de página, se reanuda la llamada al sistema completándose el proceso de **copia de memoria** y con ello la llamada, retornando **S** a modo usuario.
10. **S** reanuda su ejecución en modo usuario.

En total se producen dos copias de memoria: del *buffer* de emisión al del sistema, y de éste al de recepción. Nótese que se trata de una comunicación asíncrona donde la operación de recepción es bloqueante. El modo de operación de los sockets cuando se especifica un dominio local (`PF_LOCAL`) tiene estas mismas características.

c) Esta nueva versión intenta optimizar el caso de que el proceso receptor esté ya esperando un mensaje en el momento en el que el emisor invoca el servicio de envío. En ese caso, parece razonable ahorrarse una copia de memoria haciendo que el proceso emisor copie directamente los datos al *buffer* del receptor, eliminando el paso a través del *buffer* del sistema.
Sin embargo, hay un error en el diseño de esta optimización. Cuando está en ejecución el proceso emisor, está instalado su mapa de usuario y, por tanto, la dirección del mapa de recepción (`dir_destino` en el código planteado) se interpreta en el ámbito de este mapa, en lugar de hacerlo en el mapa del proceso receptor, que es donde tiene sentido dicha dirección. En consecuencia, los datos acabarán copiándose en el mapa del emisor, o dando un error en caso de que esa dirección no sea válida en el ámbito de dicho proceso.

d1) Básicamente, esta nueva versión produce cambios en los instantes de la traza de ejecución descrita previamente en los que se realizan copias de memoria, es decir, la primera en el punto 5, y la segunda repartida entre los puntos del 7 al 9:

5. **C** en modo sistema marca la página que contiene los datos a enviar como no expulsable y almacena en la estructura de control del puerto el número de marco asociado a esa página. Se ha eliminado, por tanto, la necesidad de la copia. Obsérvese que no ha sido necesario en este caso traer a memoria física la página que contiene el mensaje, puesto que ya está residente. El resto de este paso de la traza es igual que en la versión previa: se desbloquea al receptor y se activa la interrupción software.
7. El único cambio en este punto es que **S** comienza la copia de datos desde el marco indicado en la estructura de control del puerto, en vez de hacerlo desde un *buffer* del sistema asociado al puerto. En ambas versiones, el destino es el mismo, por lo que se produce un fallo de página que se anida con la llamada al sistema.
8. Igual que en la versión previa.
9. Al finalizar el tratamiento del fallo de página, se reanuda la llamada al sistema completándose el proceso de **copia de memoria**. A continuación, se marca la página como expulsable y se desasocia el marco copiado del puerto, terminando con ello la llamada y retornando **S** a modo usuario.

En esta versión sólo hay una copia de memoria: del *buffer* de emisión al de recepción, eliminando la necesidad de que el mensaje pase por un *buffer* del sistema.

d2) Dado que el receptor recoge directamente los datos del *buffer* del emisor, si éste modifica ese *buffer* después del envío pero antes de que el receptor obtenga los datos, estas modificaciones posteriores serían visibles en el proceso destino, lo que es claramente erróneo.
Una posible solución sería modificar la llamada `enviar` de manera que, además de marcar las páginas del *buffer* como no expulsables, las identifique como de tipo *copy-on-write* (COW). De esta forma, si el remitente modifica alguna de las páginas del mensaje después de enviarlo, se produce un fallo de tipo COW, duplicándose la página y obteniendo su propia copia, con lo que el mensaje no se ve afectado.
Por su parte, en la llamada `recibir`, después de realizar la copia, además de marcar las páginas como nuevamente expulsables, se eliminaría al COW de las mismas.
Nótese que si se producen fallos de tipo COW en todas las páginas del mensaje debido a modificaciones por parte del remitente antes de completarse la recepción, se vuelve a una situación de dos copias de memoria. Sin embargo, en la mayoría de las ocasiones, el receptor recogerá el mensaje antes de que el emisor lo modifique, por lo que sólo habrá una copia.

d3) Dado que con esta versión en el proceso de emisión de un mensaje el sistema operativo guarda la referencia de los marcos de página que contienen el mensaje, hay que asegurarse de que esos marcos de página sólo contienen datos del mensaje. Para ello, el *buffer* de usuario debe estar alineado al comienzo de una página y ocupar un número entero de páginas.
Una forma de ocultar estas restricciones es proporcionar una función que permita reservar un *buffer* que cumpla dichas restricciones. Un programa que pretenda enviar un mensaje debería usar esa función para reservar el *buffer* para el mensaje:

```
mens = reservar_buffer(sizeof(mens));
enviar(puerto, mens, sizeof(mens));
.........
liberar_buffer(mens);
```

La función que reserva el *buffer* puede crear un región de tipo anónima dentro del mapa de memoria del proceso emisor (podría ser directamente un `mmap`; y liberar un `munmap`) para asegurarse de que el *buffer* ocupa un número entero de páginas.

d4) Para conseguir una versión donde no haya copias del mensaje, es necesario eliminar en la última versión planteada la operación de copia del mensaje al *buffer* destino. Habría distintas posibilidades para lograr este comportamiento. A continuación, se describe una de ellas.
La idea está basada en que el receptor no especifica un *buffer* para obtener el mensaje, sino que es el propio sistema operativo el que le devuelve la dirección de una zona donde estará almacenado el mensaje. Por tanto, la interfaz del servicio `recibir` cambiaría. Por un lado, no habría que especificar el tamaño. Por otro lado, habría que recibir la dirección del mensaje ya sea

haciendo que el parámetro correspondiente al mensaje tuviera un nivel de indirección (en vez de ser `int recibir(int, char *, int)` sería `int recibir(int, char **)`) o bien devolviéndolo como valor de retorno (`char *recibir(int)`).

La función `recibir`, en vez de copiar los datos al *buffer* destino, habilita/crea dentro del mapa de memoria del proceso receptor, sin que éste lo solicite, una región de memoria que estaría asociada a los mismos marcos de página que el mensaje del emisor. Se podría interpretar que la comunicación acaba convirtiéndose en el establecimiento, de manera transparente a los procesos involucrados, de una región compartida de tipo COW entre el emisor y el receptor. Nótese que en esta nueva versión es necesario que el receptor notifique al sistema operativo cuando ha procesado el mensaje para eliminar el COW de las páginas asociadas al mismo (`liberar_buffer`). El modo de operación de las aplicaciones que usan este mecanismo de comunicación sería el siguiente:

Emisor

```
mens = reservar_buffer(sizeof(mens));
while(!fin) {
  generar_datos;
  rellenar mensaje;
  enviar(puerto, mens, sizeof(mens));
  .........
}
liberar_buffer(mens);
```

Receptor

```
while(!fin) {
  mens = recibir(puerto);
  procesar_datos del mensaje;
  liberar_buffer(mens);
  .........
}
```

Como se analizó para la versión previa, si el emisor modifica el *buffer* del mensaje antes de que el receptor lo procese, se producirán fallos de tipo COW, lo que rompería el compromiso del *zero-copy*. Una alternativa para eliminar este problema consistiría en que la función `enviar`, después de asociar los marcos de página con el puerto, eliminara la región del mapa del proceso emisor, desapareciendo la necesidad del COW. De esta forma, en vez de compartir con COW de forma implícita una región, la comunicación se convierte en "mover" una región entre los mapas de los dos procesos. Con esta alternativa, el emisor debería reservar un *buffer* para cada transmisión, por lo que el modo de operación resultante sería:

Emisor

```
while(!fin) {
  mens = reservar_buffer(sizeof(mens));
  generar_datos;
  rellenar mensaje;
  enviar(puerto, mens, sizeof(mens));
  .........
}
```

Receptor

```
while(!fin) {
  mens = recibir(puerto);
  procesar_datos del mensaje;
  liberar_buffer(mens);
  .........
}
```