Diseño de sistemas operativos. Junio de 2002. Ejercicio 1

Enunciado

Responda razonadamente a las siguientes preguntas sobre gestión de procesos y memoria.

1) Se plantean dos cuestiones sobre el uso de las interrupciones software por parte del sistema operativo**:**

a) Explique cómo se usa este mecanismo para llevar a cabo los cambios de contexto y qué ventajas y desventajas tiene su utilización frente a realizarlos de forma directa.

b) Un uso adicional de este mecanismo está vinculado al tratamiento de las interrupciones de los dispositivos. Explique cómo se usa este mecanismo para esta labor y qué ventajas presenta sobre el tratamiento convencional.

2) Analice cuál sería el comportamiento de la llamada al sistema `fork` en un sistema operativo con *threads* respondiendo a las siguientes cuestiones:

a) Explique qué ocurriría cuando un *thread* realiza esta llamada.

b) Detalle qué modificaciones habría que hacer en la implementación convencional del `fork` (correspondiente a un sistema sin *threads*) para adaptarlo a un sistema con *threads*.

3) Responda a las dos cuestiones del apartado anterior para la llamada `exec` en lugar de `fork`.

4) Se plantea incluir *threads* en un sistema operativo que no los proporciona usando una implementación basada en procesos de peso variable.

a) ¿Qué ventajas y desventajas presenta este estrategia frente a la implementación convencional de los *threads*?

b) Supóngase un programa que crea un proceso usando `fork`, tal que, justo a continuación, tanto el proceso padre como el hijo crean dos *threads*. Para cada una de las alternativas analizadas (implementación basada en procesos de peso variable frente a la convencional), muestre qué estructuras usa el sistema operativo para gestionar los procesos y *threads* generados por el programa.

5) Supóngase un programa que proyecta un fichero de forma privada y que, a continuación, crea un proceso mediante `fork` y este proceso hijo, a su vez, crea un *thread*. Analice, de forma independiente, qué ocurriría en las siguientes situaciones:

a) El *thread* modifica una determinada posición de memoria asociada al fichero proyectado, luego la modifica el proceso hijo y, por último, el padre.

b) El *thread* desproyecta el fichero y, a continuación, intentan acceder a esa región el proceso hijo y el padre.

Solución

1) Las interrupciones software son un mecanismo disponible en algunos procesadores que consiste en una instrucción especial, que sólo puede ejecutarse en modo privilegiado, que causa una interrupción de mínima prioridad. A continuación, se analizan las dos cuestiones planteadas en el enunciado, que se corresponden con dos usos típicos de este mecanismo.

a) Este mecanismo se suele usar para realizar los cambios de contexto de tipo involuntario. Este tipo de cambios de contexto ocurre cuando, durante la ejecución de una llamada al sistema o una interrupción de un dispositivo, se produce una situación en la que el proceso en ejecución debe dejar el procesador a otro proceso en estado de listo para ejecutar, aunque el mismo siga estando listo para ejecutar.

La principal dificultad de este tipo de cambios es que pueden producirse asíncronamente, mientras el proceso en ejecución está en modo sistema. Por tanto, la realización directa del cambio de contexto puede causar que se interrumpa una llamada al sistema o la ejecución de una rutina de interrupción sin terminar su trabajo, lo que puede causar problemas de sincronización dentro del propio sistema operativo.

El uso de las interrupciones software permite diferir el cambio de contexto involuntario hasta que haya terminado toda la actividad actual del sistema operativo. Para ello, en lugar de hacer directamente el cambio de contexto, se activa la interrupción software. Dado que esta interrupción tiene la mínima prioridad, sólo se tratará cuando haya terminado todo el trabajo actual del sistema operativo: justo antes de que se vuelva a modo usuario. Dentro de la rutina de tratamiento de la interrupción software se realizará el cambio de contexto pendiente.

Como se comentó previamente, la principal ventaja del uso de este mecanismo para realizar los cambios de contexto involuntarios es simplificar el código del sistema operativo en lo que se refiere a su sincronizaciíon interna, ya que evita la posibilidad de que haya concurrencia entre llamadas al sistema.

En el lado negativo, esta estrategia de cambio de contexto diferido no es adecuada para sistemas de tiempo real, en los que se requiere que cuando se desbloquea un proceso de alta prioridad, desaloje inmediatamente al proceso en ejecución. Asimismo, esta solución no es aplicable a multiprocesadores, donde diferir el cambio de contexto no alivia los problemas de sincronización ya que, en este caso, siempre puede haber llamadas al sistema ejecutándose en paralelo.

b) Otro uso adicional de las interrupciones software es el aplazamiento de algunas de las operaciones vinculadas con una interrupción. Hay que tener en cuenta que mientras se ejecuta una rutina de interrupción, están inhibidas todas las interrupciones de ese nivel de prioridad y de niveles inferiores. Es importante, por tanto, minimizar el tiempo que dura la rutina de tratamiento y, para ello, se puede hacer uso del mecanismo de la interrupción software.

La estrategia consiste en dejar dentro de la rutina de interrupción sólo aquellas operaciones que son realmente urgentes, mientras que el resto se difieren para que sean ejecutadas en un contexto en el que las interrupciones estén permitidas. Para ello, la rutina de tratamiento, una vez realizadas las operaciones urgentes, activa la interrupción software y termina. Será en el tratamiento de la interrupción software cuando se lleven a cabo las operaciones pendientes. La principal ventaja, por tanto, es reducir al mínimo el tiempo en el que las interrupciones están inhihidas, consiguiendo con ello un sistema con mejor tiempo de respuesta y más interactivo.

2) La llamada `fork` en un sistema UNIX convencional (sin *threads*) implica crear un proceso que es un duplicado del proceso que la invoca: básicamente, hay que duplicar el BCP y el mapa del proceso (realmente, el mapa no se copia gracias al uso de la técnica del *copy-on-write*). En un sistema con *threads* se debe mantener esta semántica. A continuación, se analiza cómo se verá afectada esta llamada cuando se "trasplanta" a un sistema con *threads*.

a) Dado que el proceso hijo es un duplicado del padre, la aplicación directa de esta semántica a un sistema con *threads* lleva a que el nuevo proceso tenga los mismos *threads* que el proceso padre, tal que cada uno de ellos es una copia del *thread* correspondiente del padre y se encuentra, por tanto, en su mismo estado y situación.

Así, cuando un *thread* `T` de un proceso invoca a `fork`, se creará un proceso hijo que tiene los mismos *threads* que el padre, entre ellos una copia del *thread* `T` que empezará a ejecutar justo después de la llamda `fork`, que, por cierto, le devolverá un 0.

Aunque no siga exactamente la semántica de crear un duplicado del padre, sería admisible una implementación alternativa en la que el proceso hijo sólo tuviera un flujo de ejecución inicial (el *thread* implícito inicial), que sea una copia del *thread* que hace la llamada.

b) La implementación convencional del `fork` debería modificarse de manera que, además de copiar el BCP y el mapa de memoria, se copien los BCTs de todos los *threads* del proceso padre.

Por lo que se refiere a la solución alternativa, habría que copiar el BCP y el mapa del padre, pero sólo el BCT del *thread* que realizó la llamada `fork`, que se convertiría en el único *thread* (el *thread* implícito) de este nuevo proceso.

3) La llamada `exec` en un sistema UNIX convencional (sin *threads*) implica que un proceso comience a ejecutar un nuevo programa. Esto implica liberar el mapa de memoria actual y crear uno nuevo vinculado con el ejecutable del nuevo programa. En un sistema con *threads* se debe mantener esta semántica. A continuación, se analiza cómo se verá afectada esta llamada cuando se "trasplanta" a un sistema con *threads*.

a) El aspecto clave que hay que entender en este caso es que si se cambia el mapa de memoria de un proceso no es posible que puedan seguir ejecutando los *threads* de ese proceso. Por tanto, cuando un *thread* invoca la llamada al sistema `exec` deben destruirse todos los *threads* de ese proceso, creándose un único flujo de ejecución inicial (el *thread* implícito inicial) que ejecutaría el código del muevo programa.

b) La implementación convencional del `exec` debería modificarse de manera que se terminarán implícitamente los *threads* liberando sus BCTs, para, a continuación, crear el *thread* implícito vinculado al nuevo programa. Realmente, no sería estrictamente necesario liberar todos los BCTs, ya que uno de ellos puede usarse para el *thread* implícito.

4) Los procesos de peso variable constituyen una modificación sobre el diseño original de UNIX. Se trata de una extensión de la llamada `fork` que permite controlar qué recursos comparten el padre y el hijo y cuáles son realmente un duplicado. Esta extensión constituye una forma alternativa de implementar *threads* en un sistema frente a la implementación más convencional. A continuación, respondiendo a las preguntas del enunciado, se analizan comparativamente estas alternativas.

a) La principal ventaja de implementar *threads* mediante el uso de procesos de peso variable es que simplifica notablemente las modificaciones que hay que realizar en el sistema operativo para incluirlos. No hay que introducir un nuevo concepto ni las estructuras de datos que conlleva (BCTs, listas de *threads* en vez de listas de procesos, planificación de *threads* en lugar de procesos, etc.). Se mantiene sólo el comcepto de proceso pero con la posibilidad de especificar el grado de compartimiento entre los procesos.

La desventaja es que con esta solución hay un mayor gasto de recursos. Para hacer patente este inconveniente, téngase en cuenta que en un sistema convencional de *threads* la información que comparten los *threads* del mismo proceso se almacena en el BCP del mismo. Sin embargo, en la solución basada en procesos de peso variable, esta información compartida será refernciada desde los BCPs de los prcesos que la comparten gastándose, al menos, un espacio de memoria adicional: los punteros que hay en los BCPs para referenciar a dicha información.

b) En el caso de un sistema convencional de *threads*, el sistema operativo usa las siguientes estructuras de datos para almacenar la información de los procesos y *threads* gebnerados en el ejemplo planteado en el enunciado:

- BCP para el proceso padre que referencia a 3 BCTs, que corresponden con el *thread* implícito y los dos *threads* creados posteriormente.
- BCP para el proceso hijo que también referencia a 3 BCTs, que corresponden con el *thread* implícito y los dos *threads* creados posteriormente.

En cambio, en la implementación basada en procesos de peso variable, las estructuras serán:

- BCP para el proceso padre.
- BCP para el primer *thread* creado por el proceso padre. Comparte recursos (mapa de memoria, ficheros abiertos, etc.) con el BCP anterior.
- BCP para el segundo *thread* creado por el proceso padre. Comparte recursos con los BCPs anteriores.
- BCP para el proceso hijo. No comparte recursos con los BCPs anteriores. Contiene un duplicado de la información existente en el BCP del padre cuando creó al hijo.
- BCP para el primer *thread* creado por el proceso hijo. Comparte recursos con el BCP anterior.
- BCP para el segundo *thread* creado por el proceso hijo. Comparte recursos con los dos BCPs anteriores.

5) Para resolver la cuestión planteada, hay que tener en cuenta, a priori, dos aspectos importantes:

- Todos los *threads* del mismo proceso comparten el mapa de memoria. Por tanto, si se elimina una región del

mapa de un proceso, esta quedará inaccesible para todos los *threads* del mismo.
- Si se proyecta un fichero de forma privada y se realiza a continuación una llamada `fork`, el padre y el hijo tienen una visión independiente del fichero. La región correspondiente no se comparte. Es una copia independiente, utilizándose la técnica del *copy-on-write* (COW) para evitar realizar una copia explícita a priori. Asimismo, hay que tener en cuenta que, al tratarse de una proyección de tipo privado, las modificaciones que se hagan sobre la región correspondiente no afectarán al fichero proyectado.

Dadas esas premisas, se responde a continuación a lo planteado en el enunciado.

a) En el escenario propuesto, el proceso padre y el hijo tendrán esa región en su mapa pero marcada como COW con un contador de 2 referencias, de manera que trabajen con copias independientes (es importante notar que no hay 3 referencias ya que el *thread* "vive" en el mapa de memoria del proceso hijo). La secuencia planteada generará los siguientes eventos:

1. Cuando el *thread* modifica una página de la región, se produce la excepción vinculada con el COW, que hace que se cree en el mapa del proceso hijo una copia propia de la página y se decremente el contador de COW.
2. Cuando el proceso hijo modifica esa página, no se produce ninguna excepción ya que el proceso hijo ya tiene su propia copia.
3. Cuando el padre la modifica, se produce la excepción del COW que, al detectar que el contador de referencias es igual a 1, no realiza la copia, sino que simplemente desactiva el COW en esa página.

Como se puede observar, las modificaciones realizas por el *thread* son visibles por el proceso hijo y viceversa. Sin embargo, los cambios realizados por el *thread* o por el proceso hijo no son visibles por el proceso padre, ni viceversa. Asimismo, hay que resaltar que ninguno de estos cambios quedarán reflejados en el fichero, que se mantendrá inalterado, al tratarse de una proyección privada.

b) Cuando el *thread* desproyecta el fichero, desaparece la región correspondiente del mapa del proceso hijo. Por tanto, cuando el proceso hijo intenta acceder a la misma se produce una excepción por acceso inválido (el "clásico" *Segmentation Fault*). El proceso padre, sin embargo, no se ve afectado y puede, por tanto, acceder a la región sin ningún problema.