1. ¿Para qué sirven los semáforos contadores? Dé al menos un ejemplo donde se aprecia su utilidad (no código).  
     
   Los semáforos contadores son utilizados para indicar la cantidad de recursos libres o disponibles que se poseen. Cuando un semáforo contador llega a cero, indica que no se poseen más recursos libres, y los procesos que intenten decrementar ese semáforo quedan bloqueados hasta que el contador sea mayor que cero.   
     
   Un ejemplo de su utilización podría ser en un sistema donde haya varias impresoras (que no implementen colas de impresión), y varios procesos intentando utilizarlas. El semáforo contador indicará las impresoras disponibles. Si un proceso está usando una impresora, se decrementará el semáforo. Cuando termine de usarla, se incrementará.
2. Un procesador recibe un trap, y por ende el S.O comenzará la ejecución de la RAI asociada. Si durante esa ejecución llegara una nueva interrupción de hardware, ¿podemos decir que habrá context switch? ¿Hay cambio de modo en ese caso? ¿Por qué?  
     
   <https://stackoverflow.com/questions/24128926/what-happens-when-an-isr-is-running-and-another-interrupt-happens>   
     
   Sí. Una interrupción puede interrumpir la RAI de otra interrupción, si esta nueva es de mayor prioridad que la anterior. Habrá un cambio de contexto, ya que el procesador debe guardar el entorno de ejecución de la RAI que ha sido interrumpida, y preparar los registros para ejecutar la RAI de la interrupción de mayor prioridad.  
     
   No habrá un cambio de modo, ya que todo sucede en el kernel space. No se necesita cambiar de modo usuario a modo kernel.
3. Un sistema operativo multiusuario, multitarea que ejecuta los procesos con ULT e implementa Jacketing, se encuentra ejecutando un proceso que está dividido en 4 threads. El estado actual de tres de los thread es Ready, y el restante es Running. En un determinado momento el thread en ejecución ejecuta un syscall para acceder a un archivo en el file system.   
   Se pide que seleccione la/s opciones que corresponda y ordene la secuencia de los eventos seleccionados.   
   Por cada evento se deberá indicar si el mismo produce (sí), no se produce (no), o depende de alguna condición adicional (explicar claramente cuál es ese condicionante).  
   En caso de existir más de una opción, tachar lo que no corresponda (indicado todo en mayúsculas).

|  |  |  |
| --- | --- | --- |
| **Orden** | **Sí/No/Depende** | **Evento** |
| 3 | Depende | El proceso en Running pasa a la cola de SUSPENDIDOS / LISTOS / FINALIZADOS |
|  | No | Se produce una interrupción ENMASCARABLE / NO ENMASCARABLE |
|  | No | Se invoca una RAI |
|  | No | Se ejecuta una RAI que es en modo KERNEL / USER |
| 2 | Sí | El proceso continúa en estado running |
|  | No | Se activa el FLIH |
|  | No | Se inicia un context switch |
| 1 | Sí | Se produce una syscall |
|  | No | Se liberan los recursos asignados a ese proceso |
|  | No | Ejecuta el dispatcher |
|  | No | Ejecuta el traffic controller |
|  | No | Se dispara una interrupción por SOFTWARE / HARDWARE INTERNA / EXTERNA |
|  | No | El thread en ejecución se bloquea y se asigna otro thread del mismo proceso |

1. En un concurso de matemáticas se presentó el desafío de elaborar un software para realizar complejos mapas de contaminación acústica, basados en curvas de nivel (puro CPU Bound). Tres grupos tomaron el desafío y tiempo después presentaron su solución. Las tres fueron exitosas y cumplieron correctamente su cometido. Sin embargo, un ingeniero notó importantes diferencias en el rendimiento de cada desarrollo. El primero de ellos fue construido utilizando hilos ULT (main thread + 4 hilos), el segundo software se construyó con un único programa sin creación de hijos ni hilos, y el tercero sin hilos, pero usando tres procesado pesados cooperativos. En todos los casos se utilizó el lenguaje C y se probaron sobre el mismo equipo, con un sistema operativo de propósito general monoprocesador y con soporte para multiprogramación.   
     
   ¿Puede determinar cuál de los tres productos ofrece el mejor rendimiento y por qué motivo?  
     
   El producto que realizó la separación en hilos ULT. Al usar hilos ULT, se reduce la cantidad de overhead que se experimenta al hacer process switch en la tercera solución (ya que en esta tenemos que estar constantemente cambiando entre los procesos cooperativos para poder progresar).   
     
   Además, es posible que el producto hecho con hilos ULT haya adoptado una estructura de hilos que trabajen en equipo. La estructura en equipo es generalmente utilizada para resolver cuestiones de cálculo, ya que tiene un mejor rendimiento. Esta sería la ventaja que el producto de hilos ULT tendría sobre el que se realizó en un solo proceso sin hilos.   
     
   También, al ser hilos ULT, es el programador el que decide la planificación de los hilos… y si esto se hace bien se podría tener un resultado más eficiente que siendo un programa sin hilos.
2. Dos procesos pueden sincronizarse utilizando semáforos siempre y cuando corran sobre el mismo procesador, en cualquier otro caso deberán sincronizarse mediante el uso de otros mecanismos.  
     
   Falso. Los semáforos son almacenados en memoria principal y no tienen ninguna relación directa con el procesador en el que se está ejecutando el proceso.
3. Se tiene un sistema operativo monoprocesador y multitarea. En dicho sistema se ejecuta todas las noches un proceso que realiza 5 backups de distintos orígenes de datos (de una BD, de un sistema de archivos, etc). Pero todos esos backups se realizan sobre una única unidad de cinta.   
   El proceso es completamente secuencial (se hacen los backups de a uno) y tarda varias horas en terminar.  
   ¿Qué estrategia propondría para mejorar ese tiempo? ¿Se podría dividir ese proceso en varios? ¿Y si en lugar de dividirlo en procesos se hace con threads? Analice la situación y elabore una conclusión.   
     
   Hay un claro cuello de botella provocado por el hecho de que este proceso es I/O bound, y que, además, se escribe todo en la misma unidad de cinta. No contamos con varias unidades de cintas, por lo que no podemos aprovechar y escribir en paralelo…  
     
   El proceso podría dividir en varios, y de esta forma por lo menos la lectura de los archivos y los registros sería más rápida (la escritura se mantendría igual). Sin embargo, se genera un gran costo debido a la sobrecarga (overhead) que causaría tener que estar haciendo tantos process switchs.  
     
   Si implementásemos hilos ULT, podríamos reducir el overhead, pero bloquearíamos todos los hilos cada vez que realizásemos una syscall. Podríamos utilizar jacketing, pero el jacketing involucra re-escribir todas las bibliotecas para que no sean bloqueantes, y esto es ineficiente.   
     
   Si implementásemos hilos KLT, tendríamos el mismo overhead, pero en caso de que se bloquee un hilo podemos ejecutar otro.  
     
   Mi recomendación sería dividirlos en varios hilos KLT. Los hilos tienen la ventaja de compartir el mismo espacio de memoria, por lo que gastaríamos menos recursos que creando varios procesos pesados. Esto más lo anteriormente dicho, para mí, daría un aumento de eficiencia.
4. Suponga que en un sistema non preemtive hay 3 procesos (A, B, y C). En un momento dado, A se encuentra ejecutando, Y B y C están en estado “Bloqueado”. Unos instantes después llega una interrupción de software. Seleccione la opción válida y justifique su respuesta.   
     
   El S.O pasará A a Listos, atenderá la interrupción y luego le dará el turno nuevamente a A, ya que los otros dos están bloqueados.   
   El S.O primero atenderá la interrupción, pero A conservará su estado “Running”.  
   [ X ] El S.O. no podrá atender la interrupción hasta que A libere la CPU, porque el sistema es non-preemptive.   
   Ninguna de las afirmaciones es correcta. Si elige esta opción explique qué ocurrirá.  
     
   En los sistemas non-preemtive, los procesos “cooperan” para compartir el uso del procesador. Es decir, son ellos los que deciden cuando detener su ejecución y otorgarle el uso del procesador a otro proceso o al sistema operativo. El sistema operativo nunca realiza un cambio de contexto.
5. Si se ha producido un trap de división por cero, este puede ser capturado en la programación por una instrucción Try/Catch (o equivalente) para evitar que finalice el proceso y continué con el flujo normal de ejecución.   
     
   Verdadero. Es una parte del manejo de excepciones de hardware interno. Los try catchs cumplen con la función de “intentar” ejecutar una instrucción y “capturar” la excepción, en caso de que se produzca una. De esta forma se puede continuar con el flujo normal del proceso sin que el mismo termine su ejecución.
6. Un pipe “|” es un buffer en memoria que funciona como FIFO en el que un proceso escribe y el otro lee.   
     
   Verdadero. Los pipes sin nombre son un área de memoria que son tratados como un “archivo virtual”. En sí, son un buffer.
7. ¿Se puede producir deadlock en un sistema en que los recursos involucrados estén administrados por un spool?   
     
   No, no se puede. El spool virtualiza los recursos, es decir, permite que un recurso pueda ser utilizado por varios procesos sin tener que esperar a que otro proceso lo libere. De esta forma, se rompe la condición de mutua exclusión, una de las cuatro condiciones necesarias de Coffman para que se produzca deadlock.
8. Si se produce una excepción del tipo “segmentation-fault” en un sistema con memoria virtual, significa que el proceso se encuentra en Listos-suspendidos o Bloqueados-suspendidos.   
     
   Falso. Si se produce una violación de segmento, es porque el proceso quiso ejecutar una instrucción que apunta a un segmento (área) de memoria que no le pertenece. Para que suceda esto, el proceso debe estar en ejecución.