Sistemas Operativos
Control 3

2 horas

Junio de 2009

Pregunta 1 (kernel)

Parte I

Para sincronizar un mutex, se propone la siguiente implementación para un sistema con multi-procesadores:

```
#define ESTA 1
#define NO_ESTA 0

lock(int *l) {
    while(*l == NO_ESTA) /* espero */
;

    *l = NO_ESTA;
}

unlock(int *l) {
    *l = ESTA;
}
```

Discuta si es correcta y proponga una mejor implementación.

La implementación no es correcta, ya que varios procesos podrían estar esperando en el while y ver al mismo tiempo que la llave está, salir del ciclo y tomarla simultáneamente. Para evitar eso, la única solución es utilizar una operación atómica del hardware para probar y setear un valor (test and set) o un swap de dos posiciones de memoria. Si supongo una operación atómica de test and set:

```
lock(int *l) {
    while(test_and_set(l, NO_ESTA) == NO_ESTA)
;
}
```

Parte II

Un ingeniero dice que no se puede hacer un Linux seguro aceptando carga dinámica de módulos con drivers. Por otro lado, casi todas las distribuciones de Linux soportan carga dinámica. Explique las razones de esta discusión y si tienen razón.

Ambos tienen razón: la carga dinámica de módulos es un problema de seguridad, pero es demasiado útil y permite hacer sistemas flexibles y más dinámicos, de modo de agregar hardware en forma mucho más sencilla. El principal problema de los drivers en módulos es que dentro del kernel tienen derecho a hacer lo que quieran, sin ninguna protección para el resto de los procesos del sistema.

Parte III

Explique por qué es necesario hacer sincronización dentro del kernel y no basta con tener un gigantesco mutex cada vez que se entra al kernel que se libera cuando uno sale del kernel.

Tener un mutex gigante resolvería el problema, pero sería terriblemente ineficiente. El porcentaje de tiempo que un procesador pasa en el kernel es muy alto (alrededor del 50 % del tiempo) y si tenemos varios procesadores esto implica que todos estarían esperando entrar y al final terminaríamos ejecutando secuencialmente y con costos de sincronización que nos harían más lentos que un sistema mono-procesador.

Pregunta 2 (Discos)

Parte I

Para poder tener un buen algoritmo de scheduling de la E/S a disco, se requiere poder re-ordenar los accesos, cambiando el orden original. ¿Cómo puedo garantizar que el resultado (para una aplicación) será siempre el mismo a pesar que los accesos se harán en cualquier orden?

Lo más importante es tener un cache común para todo el acceso a disco, que sea compartido por todas las aplicaciones. Entonces, las escrituras y lecturas se hacen ordenadas en el cache, y su orden en disco es irrelevante. Si una aplicación escribe un dato en disco y luego le avisa a otra que ya está listo, la segunda leerá el dato correctamente aunque no se haya escrito aun a disco, puesto que lo sacará del cache.

Parte II

Explique qué significa que un sistema de archivos esté fragmentado y cómo se corrige.

En los discos se habla de fragmentación cuando los bloques de un mismo archivo se encuentran en cilindros distintos. Esto genera que las lecturas secuenciales son muy lentas puesto que debo mover la cabeza del disco para cada bloque. Se corrige moviendo los bloques para que queden en el mismo cilindro y, ojalá, contiguos.

Parte III

Cuando usamos NFS para accesar discos remotos, puedo tener que retransmitir peticiones de operaciones debido a fallas. Explique cómo el protocolo garantiza que el resultado será siempre el mismo.

Todas las operaciones en NFS son idem-potentes: se pueden realizar múltiples veces y generan siempre el mismo resultado. Por ejemplo, no existe la operación .ªgregar al final del archivo", sino la operación .ºescribir en la posición 2034 del archivo estos 10 bytes". En NFS, cuando el sistema está montado en "hard.º en "duro", la operación será transmitida todas las veces que sea necesario, hasta recibir la respuesta correcta.

Pregunta 3 (Redes)

Parte I

Explique en qué se diferencian los dispositivos de red del resto de los drivers estudiados.

Son muy distintos: ocupan un espacio de nombres propio (no existen en /dev), tienen operaciones que implementan diferentes, las aplicaciones no los ocupan directamente, y toda la administración se hace vía comandos ad-hoc (ifconfig).

Parte II

Explique por qué no tiene sentido que una aplicación de usuario abra directamente la red (un /dev/network) para enviar/recibir datos como un socket.

La principal razón es que un dispositivo de red se multiplexa entre varias conexiones a diversos clientes/servidores, generando un socket para cada conexión. Por lo tanto un /dev/network tendría que ser compartido entre múltiples procesos y lograr diferenciar cada conexión.

Parte III

La API oficial de un driver de red posee una función `xmit` para enviar datos a la red. Sin embargo, no tiene ninguna función para recibir datos. Explique por qué.

La red, a diferencia de otros dispositivos, genera datos por su propia cuenta, para diversos procesos que esperan en sus sockets. Por lo tanto, la lectura se realiza al atender las interrupciones de la red y no con una primitiva de read.