DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Resumen Sistemas Operativos

Axel Straminsky

Diciembre 2015

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Índice

1. Procesos

1.1. Introduccion

Un proceso es una abstracción de un programa en ejecución, que básicamente convierte una CPU en varias CPU virtuales, proporcionando de esta manera la capacidad de operar (pseudo)concurrentemente, incluso cuando hay una sola CPU disponible.

Un proceso no es más que una instancia de un programa en ejecución, incluyendo los valores actuales del contador de programa, los registros y las variables. En concepto, cada proceso tiene su propia CPU virtual; en la realidad, la CPU real conmuta de un proceso a otro. Varios procesos pueden compartir un solo procesador mediante el uso de un algoritmo de planificación para determinar cuándo se debe detener el trabajo en un proceso para dar servicio a otro.

Hay cuatro eventos principales que provocan la creación de procesos:

- El arranque del sistema
- La ejecución de una llamada al sistema para creación de procesos.
- Una petición de usuario para crear un proceso.
- El inicio de un trabajo por lotes.

Algunos procesos corren en primer plano, es decir, interactúan con los usuarios, mientras que otros procesos, que no están asociados con usuarios específicos sino con una función específica, corren en segundo plano. Un ejemplo sería un proceso que acepte peticiones entrantes para las páginas web hospedadas en ese equipo; estos procesos se conocen como **daemons**.

En UNIX sólo hay una llamada al sistema para crear un proceso: **fork**. Esta llamada crea un clon exacto del proceso que hizo la llamada. Después del fork, los dos procesos (padre e hijo) tienen la misma imagen de memoria, las mismas variables de entorno y los mismos archivos abiertos. Por lo general, el proceso hijo ejecuta después a **execve** o una llamada al sistema similar para cambiar su imagen de memoria y ejecutar un nuevo programa. La razón de este proceso de dos pasos es para permitir al hijo manipular sus descriptores de archivo después de fork, pero antes de execve, para poder lograr la redirección de la entrada, salida y error estándar.

1.2. Estados de un proceso

Un proceso se puede encontrar en uno de tres estados:

- En ejecución (está usando la CPU en ese instante)
- Listo (ejecutable. Se detuvo temporalmente para dejar que se ejecute otro proceso.)
- Bloqueado (no puede ejecutarse hasta que ocurra cierto evento externo)

1.3. Implementación de los procesos

Para implementar el modelo de procesos, el SO mantiene una tabla llamada tabla de procesos, con una entrada por cada proceso (llamada Process Control Block (PCB)). Esta entrada contiene información importante acerca del estado de cada proceso, como el Program Counter, el Stack Pointer, asignación de memoria, estado de sus archivos abiertos, información para el scheduler, registros de la CPU, etc.

Cuando ocurre una interrupción, el PC, el PSW, y uno o más registros se meten en la pila mediante el hardware de interrupción. Después, la CPU salta a la dirección especificada en el vector de interrupción. Esto es todo lo que hace el hardware, en adelante es todo responsabilidad del software.

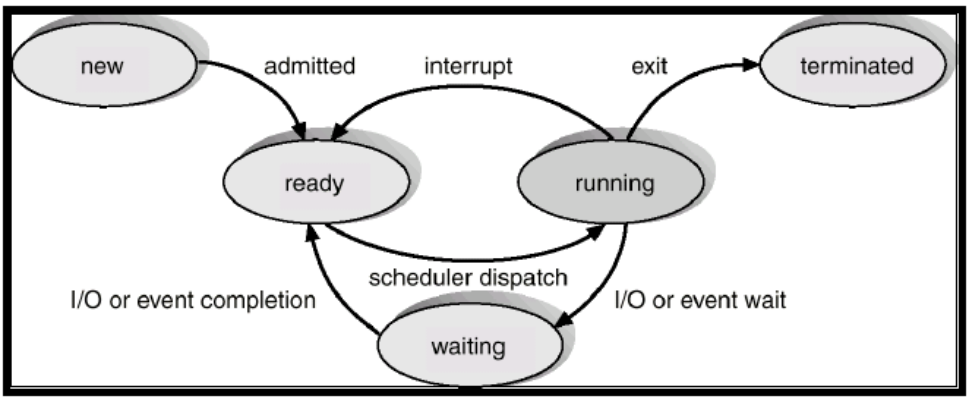

Figura 1: Estados de un proceso

Todas las interrupciones comienzan por guardar los registros, a menudo en la entrada de la tabla de procesos para el proceso actual. Después, se quita la información que la interrupción metió en la pila y el stack pointer se establece para que apunte a una pila temporal utilizada por el manejador de procesos.

Cuando se terminan de guardar los registros, se llama a un procedimiento para realizar el resto del trabajo para este tipo de interrupción específica. Cuando termina su trabajo y tal vez algún otro proceso está listo, el scheduler es llamado para ver qué proceso se debe ejecutar a continuación. Después de eso, el control pasa de vuelta al código en lenguaje ensamblador para cargar los registros y el mapa de memoria para el proceso que entonces es el actual.

Un resumen de los pasos que ejecuta el sistema operativo cuando ocurre una interrupción es el siguiente:

1) El hardware mete el PC, PSW, registros, etc. a la pila.

2) El hardware carga el nuevo PC del vector de interrupciones.

3) Procedimiento en lenguaje ensamblador guarda los registros.

4) Procedimiento en lenguaje ensamblador establece la nueva pila.

5) El servicio de interrupciones de C se ejecuta.

6) El scheduler decide qué proceso se va a ejecutar a continuación.

7) Procedimiento en C regresa al código en ensamblador.

8) Procedimiento en lenguaje ensamblador inicia el nuevo proceso actual.

A continuación un ejemplo de una interrupción en el contexto de una syscall:

1.4. Threads

Los threads son una especie de "mini roceso.º "proceso ligero"dentro de un proceso clásico. Los threads posibilitan el concepto de procesos secuenciales que realizan llamadas al sistema con bloqueo y de todas formas logran paralelismo.

Una manera de ver a un proceso es como si fuera una forma de agrupar recursos relacionados, como archivos abiertos, procesos hijos, etc., además de su espacio de direcciones. El otro concepto que tiene un proceso es el de hilo de ejecución, el cual tiene un contador de programa, registros con sus variables actuales y una pila. Aunque un hilo se deba ejecutar en cierto proceso, el hilo y su proceso son conceptos distintos y pueden tratarse por separado.

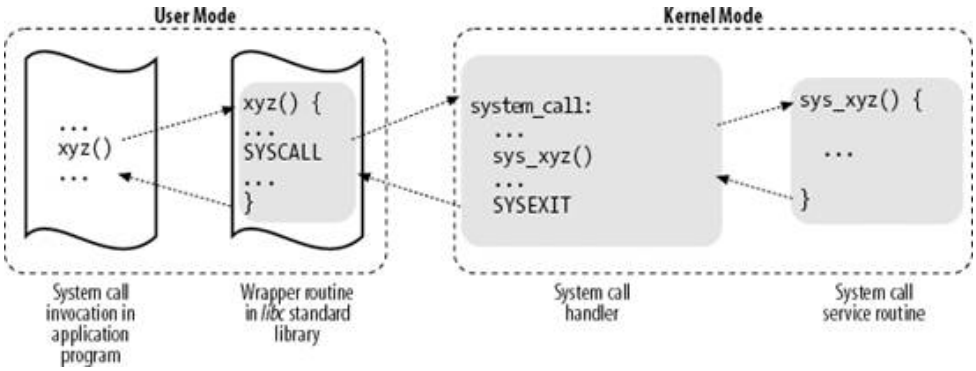

Figura 2: syscall

Lo que agregan los hilos al modelo de procesos es permitir que se lleven a cabo varias ejecuciones en el mismo entorno del proceso, que son en gram parte independientes, por lo tanto, los hilos dentro de un proceso comparten el mismo espacio de direcciones. No hay protección entre los hilos debido a que (1) es imposible y (2) no debe ser necesario. A diferencia de tener procesos diferentes, que pueden ser de distintos usuarios y hostiles entre sí, un proceso siempre es propiedad de un solo usuario, quien se supone ha creado varios hilos para que cooperen. Además de compartir un espacio de direcciones, todos los threads pueden compartir el mismo conjunto de archivos abiertos, procesos hijos, señales, etc. Al igual que un proceso tradicional, un thread puede estar en uno de varios estados: ene ejecución, bloqueado, listo o terminado.

Hay 2 formas principales de implementar una librería de threads: en espacio de usuario o en el kernel, aunque también es posible una implementación híbrida.

1.4.1. Implementación de threads en espacio de usuario

En este caso la librería de threads reside completamente en espacio de usuario, y el kernel no sabe nada de ellos. En lo que al kernel concierne, está administrando procesos ordinarios con un solo thread.

En este caso, cada proceso tiene su propia tabla de threads. Algunas ventajas son que cuando un thread se bloquea y se debe conmutar con otro, el procedimiento es mucho más rápido que hacer el trap al kernel. Otra ventaja es que cada proceso puede tener su propio algoritmo de planificación personalizado.

A pesar de su buen rendimiento, los threads a nivel de usuario tienen algunos problemas importantes. El primero de todos es la manera en que se implementan las llamadas al sistema de bloqueo. Si, por ejemplo, un thread lee del teclado antes de que se haya oprimido una sola tecla, es inaceptable permitir que el thread se bloquée, ya que esto también bloqueará a todos los threads del proceso. Si un thread produce un fallo de página, el kernel naturalmente bloquea a todo el proceso, llevando al mismo problema que en el caso anterior.

Otro problema es que si un thread comienza a ejecutarse, ningún otro thread en ese proceso se ejecutará a menos que el primero renuncie de manera voluntaria a la CPU, ya que dentro de un proceso no hay interrupciones.

1.4.2. Implementación de threads en el kernel

En este caso, en vez de tener una tabla de threads por proceso, el kernel tiene una tabla de threads que lleva la cuenta de todos los threads del sistema. Además, el kernel mantiene la tabla de procesos tradicional. Cuando un proceso desea crear un nuevo thread, realiza una llamada al kernel. Los threads del kernel no requieren de nuevas llamadas al sistema no bloqueantes, y si se produce un fallo de página, el kernel puede comprobar con facilidad si el proceso tiene otros threads que puedan ejecutarse y los ejecuta. Su principal desventaja es que el costo de una llamada al sistema es considerable. Sin embargo, hay algunos problemas que no resuelven, como qué hacer luego de un fork, o qué hilo debe hacerse cargo de una señal enviada al proceso.

1.4.3. Implementaciones híbridas

Una manera de combinar los métodos anteriores es utilizar threads de nivel kernel y después multiplexar los threads de nivel usuario con alguno o todos los threads de nivel kernel. El programador puede determinar cuántos hilos de kernel va a utilizar y cuántos threads de nivel usuario va a multiplexar en cada uno.

1.5. Sincronización entre procesos

Los bugs que surgen cuando dos o más procesos están leyendo o escribiendo en algún buffer compartido y el resultado final depende de quién se ejecuta y exactamente cuándo lo hace, se conocen como **condiciones de carrera.**

Para evitar esto, se necesita **exclusión mutua**, es decir, cierta forma de asegurar que si un proceso está utilizando una variable o archivo compartido, los demás procesos se excluirán de hacer lo mismo. La parte del programa en la que se accede a la memoria compartida se conoce como **región crítica.**

Para solucionar esto, se busca cumplir las siguientes cuatro condiciones:

- No puede haber dos procesos de manera simultánea dentro de sus regiones críticas.
- No pueden hacerse suposiciones acerca de las velocidades o el número de CPUs.
- Ningún proceso que se ejecute fuera de su región crítica puede bloquear otros procesos.
- Ningún proceso tiene que esperar para siempre para entrar a su región crítica.

Algunas proposiciones para lograr exclusión mutua son las siguientes:

- **Deshabilitar interrupciones:** en un sistema con un solo procesador, la solución más simple es hacer que cada proceso deshabilite todas las interrupciones justo después de entrar a su región crítica y las rehabilite justo después de salir. A pesar de que esto soluciona el problema de las condiciones de carrera, introduce un nuevo problema, ya que si el proceso no rehabilita las interrupciones, el sistema no podría continuar ejecutándose.

  En un procesador multinúcleo, al deshabilitar las interrupciones de una CPU no se evita que las demás CPUs interfieran con las operaciones que la primera CPU está realizando. En consecuencia, se requieren esquemas más sofisticados.

- **Variables candado:** es una solución por software que consiste en tener una variable compartida que está en 0 cuando no hay ningún proceso en su región crítica y 1 cuando si lo hay. Este método tiene el mismo problema que el anterior, ya que si 2 procesos leen el 0 a la vez, ambos fijarán el candado en 1 y entrarán en su región crítica.

- **Alternancia estricta:**

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region();
    turn = 1;
    noncritical_region();
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region();
    turn = 0;
    noncritical_region();
}
```

(b)

Cuando un proceso está evaluando continuamente una variable (como *turn*), se dice que está haciendo **busy waiting**.

El problema con este enfoque es que si uno de los procesos es mucho más lento que el otro, este otro proceso quedará haciendo busy waiting hasta que el proceso más lento vuelva a cambiar la variable candado. Esta situación viola la condición 3 antes establecida: un proceso está siendo bloqueado por otro proceso que no está en su región crítica. Por más que evite las condiciones de carrera, no es un candidato serio como solución.

- **Solución de Peterson:**

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE)  /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

- **Instrucción TSL:** TSL (test-and-set-lock) es una instrucción del hardware que funciona de la siguiente manera: lee el contenido de una palabra de memoria (candado) y lo guarda en un registro, y después almacena un valor distinto de cero en la dirección de memoria del candado. Se garantiza que las operaciones de leer la palabra y almacenar un valor en ella serán indivisibles; la CPU que ejecuta la instrucción TSL bloquea el bus de memoria para impedir que otras CPUs accedan a la memoria hasta que termine. Para lograr esto, se deshabilita el bus de memoria para el resto de los procesadores. Para usar la instrucción TSL necesitamos una variable compartida (candado) que coordine el acceso a la memoria compartida. Cuando candado es 0, cualquier proceso lo puede fijar en 1 mediante el uso de la instrucción TSL y después una lectura o escritura en la memoria compartida. Cuando termina, el proceso establece candado en 0 mediante una instrucción move ordinaria.
- **sleep y wakeup:**