Arquitectura del Computador
y
Sistemas Operativos

Decimonovena Clase

Procesos (1/2)

El proceso es la abstracción de un programa ejecutando. En esa abstracción el SO guarda la siguiente información:

- El código del programa (todos sus módulos)
- Los datos generados al programar
- Los datos generados durante la ejecución

Éste es el programa

- Los recursos "*virtuales*" actualmente en uso por el programa (archivos abiertos, buffers de pantalla, semáforos, mutex, conexiones de red, etc.)
- Información de prioridad de ejecución y tiempo de ejecución
- Información de seguridad (usuario dueño del proceso, etc)
- Si el proceso no está corriendo, el estado de todos los registros

Procesos (2/2)

Toda esta información se almacena en un arreglo de estructuras llamado "*Process Table*", que el SO mantiene en un área para su exclusivo uso.

Los SOs permiten que las tareas se identifiquen a sí mismas o a otras tareas entregando una referencia a un elemento de esa tabla. Esta referencia puede llamarse Process Id (PID), Process Handle, etc.

Multiprocesamiento

Si bien la capacidad de colocar múltiples procesadores en un ordenador es muy posterior al uso de uno solo, la ejecución de múltiples procesos en forma simultánea viene desde muy al comienzo de los sistemas.

Al ser el procesador mucho más rápido que el mundo alrededor suyo (impresoras, discos, interfaces de red y por qué no los mismos usuarios), puede ejecutar pequeñas porciones de cada programa e ir saltando de uno a otro dando la "*apariencia*" que todos corren simultáneamente. La ilusión puede compararse con el cine o la televisión, donde una secuencia de imágenes estáticas son percibidas por el ser humano como un movimiento continuo.

El tiempo durante el cual un proceso ejecuta se denomina "*time slice*".

Esta metodología convierte el tiempo en "*vitual*". Ahora, si le medimos el tiempo de ejecución a un lazo que itera 10000 veces, no tarda lo mismo cada vez que lo ejecutamos, dado que cuánto le lleva terminarlo depende de qué otras tareas estén ejecutando a la vez.

Dado que un proceso no tiene cosas para hacer el 100% del tiempo, el multiprocesamiento mejora la utilización del sistema.

Vida de un Proceso - Creación (1/2)

Los SOs tienen alguna función para iniciar un proceso. Un parámetro obligado es el programa a ejecutar, que dependiendo del sistema podrá estar en disco, memoria ROM, etc.

Los procesos se crean por el solo inicio del sistema (el proceso de booting termina ejecutando un shell), o porque un proceso corriendo arranca otro (sea con intervención del usuario o sin ella).

- En Windows es más sencillo, se ejecuta la función CreateProcessAsUser() que hace casi todo el trabajo. En ella se especifica el usuario, el programa, los argumentos, variables de entorno, etc. Luego pueden ser necesarios algunos ajustes menores, como SetPriorityClass().

- En Linux es más complejo. La única forma de crear un proceso es con fork(), pero genera un clon del actual. Luego hay que llamar setuid(), setgid(), armar en entorno y por último a execve() para cambiar el programa. También pueden requerirse otras como setpriority().

Vida de un Proceso - Creación

- En Android usamos getPackageManager().getLaunchIntentForPackage() y startActivity(). Sustancialmente más simple porque por default no maneja múltiples usuarios usuarios. También se puede usar setPriority() de ser necesario.

Lo importante es entender que una vez lanzado el nuevo proceso, tiene un espacio de memoria completamente independiente Inclusive bajo Linux, que al hacer el fork() duplica el proceso actual, no hay memoria "*escribible*" compartida, todo se duplica.

Algunos SOs como Linux sí comparten ciertas zonas de memoria entre procesos, como el código del programa. Éso es porque son zonas que no se pueden modificar, así que duplicarlas es un desperdicio de recursos.

Vida de un Proceso - Terminación

En muchos sistemas operativos el proceso al terminar devuelve un “código de resultado”, que es un indicador de éxito o fracaso. Este código puede ser consultado por otro proceso para alterar su funcionamiento.

Los procesos pueden terminar por diversos motivos.

- Voluntad propia: El proceso terminó lo que tiene para hacer, o detecta condiciones tales en que ya no puede seguir adelante. El “*código de resultado*” puede indicar cuál es el caso.

- Se detectó un error fatal (el programa tiene errores o intenta hacer cosas que no están permitidas). En este caso es el SO el que lo cierra.

- Un usuario con la debida autorización decide “matar” el proceso.

En el primer caso el programa llamó voluntariamente a una función del SO diciéndole que tiene que cerrarlo y pasándole el “*código de resultado*”.

En los dos últimos casos el programa no terminó, su ejecución fue suspendida en el punto en que se tomó la decisión de terminarlo.

Vida de un Proceso - Estado

Hacer multiprocesamiento es algo más complejo que saltar de un proceso a otro. Por ejemplo si un proceso está esperando una tecla, no tiene sentido ejecutarlo hasta tanto no haya una disponible, sería un desperdicio de recursos. Por ese motivo los procesos están en distintos estados. Cada SO define los propios:

Linux	Windows
Runnable/Running	Ready to Run
	Running
Stopped	Suspended
Zombie	Ended
Interruptible sleep	Blocked/Waiting
Uninterruptible sleep	

Threads

Hasta ahora dijimos que cada proceso tiene un único puntero que indica dónde está ejecutando.

Cuando alguna de las tareas a realizar requiere una espera de un recurso externo (lento), hacerlas en forma secuencial involucra pedir y esperar muchas veces. Resulta mucho más óptimo tener varios puntos de ejecución paralelos dentro del mismo proceso, así se despachan todos los pedidos, y se espera una única vez.

Estos varios puntos de ejecución paralelos se denominan hilos. Cada hilo tiene sus propios registros y su propio stack. Sin embargo no hay restricción en cuanto al acceso a la memoria y/u otros recursos asignados al proceso al que pertenecen.

Por supuesto cada hilo o "*Thread*" debe tener una estructura en la memoria del SO, en la que se mantenga, entre otras cosas, un estado. Referencias a esta tabla, o *Thread Table*, reciben el nombre de Thread ID, o Thread Handle.

El SO ahora divide la capacidad de cómputo entre todos los threads de todos los procesos.

Context switching - Costo

Cuando es necesario "*sacar*" a una tarea de la CPU y colocar otra, es necesario hacer varias cosas. Este grupo de tareas se denomina "*Context Switching*".

Dado que los threads comparten ciertos recursos con otros threads del mismo proceso, el "*costo*" (en términos de tiempo perdido) de cambiar de hilo es mucho menor que el de cambiar de proceso.

Además, precisamente porque no comparten memoria, al cambiar de proceso se vacía completamente:

- El cache del core
- El "Translation Lookaside Buffer" o TLB (que veremos al estudiar organización de memoria)

Prioridades

No todos los procesos tienen la misma prioridad. La asignación de prioridades no está relacionada con la importancia de la tarea, sino que puede depender de varios factores:

- Si la tarea está al frente (interactuando con el usuario).
- El deseo del usuario, que puede haber subido la prioridad de un proceso para que termine antes.
- Si el proceso atiende un dispositivo externo que requiere atención periódica o tendrá fallas (grabación de DVDs)
- Si es posible determinarlo, cuánto le falta a la tarea para terminar.

Según el sistema operativo las prioridades pueden ser:

- Estáticas: Las asigna el usuario y/o el desarrollador y se mantienen durante toda la vida del proceso.
- Dinámicas: El SO las altera en función de la cantidad de CPU que usa, la carga del sistema, etc. Volveremos sobre ésto al ver “*scheduling*”.

Scheduling (1/2)

Al proceso de elegir cuál(es) hilo(s) ejecutarán a continuación se lo conoce como "*scheduling*". No es algo simple y cada sistema operativo tiene su propio algoritmo. Algunas particularidades suelen ser:

- No se puede elegir siempre la tarea de mayor prioridad, dado que la de poca prioridad nunca ejecutaría. Ésto se llama "*Starvation*".
- Que un hilo reciba el control no significa que ejecute durante todo su "time slice". Puede detenerse porque busca en recurso que no está o porque el SO la interrumpe para ejecutar otra tarea de mayor prioridad que ahora está en estado ejecutable.
- Se debe considerar de alguna manera a qué proceso corresponde cada thread, dado que sino un proceso crearía muchísimos threads y se quedaría con casi todos los recursos.
- Si una tarea de alta prioridad está esperando un recurso que tiene otra de baja prioridad deberíamos acelerar esa tarea. Ésto se conoce como "*Priority Inversion*".

Scheduling (2/2)

Hay varios tipos de algoritmos posibles:

A) Por proceso:

- Shortest-Job-First: En vez de usar prioridades, los trabajos que se sabe pueden terminarse en menos tiempo se ejecutan primero.
- First-Come-First-Serve: Las tareas se ejecutan por orden de llegada.
- Priority Scheduling: Aquí las tareas tiene prioridad, se ejecutan primero las de mayor prioridad y luego las otras. Dentro de igual prioridad se usa uno de los criterios anteriores.
- Highest Response Ratio Next: Considera no solamente el tiempo que le falta para terminar sino el que lleva esperando.

A) Por thread:

- Round Robin: Se colocan en círculo todos los threads ejecutables y se ejecuta un timeslice de cada una.
- Multilevel Feedback Queue: Se crean colas por prioridad, pero las tareas se sube y/o bajan en función de otros factores.
- Greedy algorithm: Consiste en tomar la mejor elección en cada paso esperando que éso lleve a la mejor solución total.

Collaborative Multiprocessing/Multithreading

Durante muchos años se usaron procesadores que no estaban diseñados para estas tareas. El Sistema Operativo DOS (en parte) y el Windows 3.11 (primer windows ampliamente usado) corrían sobre el procesador 8086 que no lo soportaba.

Estos sistemas basaban la multitarea en que una tarea "*cedía*" el control llamando al SO para avisarle que no tenía nada que hacer. El SO entonces le daba el control a otra tarea. Nuna era el propio SO el que iniciaba el "*task switch*", sino que el cambio se producía solamente cuando la tarea lo indicaba o porque la tarea solicitaba un recurso que no estaba disponible.

Precisamente para abordar este problema, en Windows nace el concepto de "Mensajes", que sigue hasta el día de hoy. Los procesos no tenían un lazo principal donde iteraban mientras llevaban a cabo su objetivo, sino que procesaban en respuesta a esos mensajes.

Preemptive Multiprocessing/Multithreading

Este método resuelve el problema causado por una tarea que no cede el control.

A cada hilo se le asigna un tiempo máximo de ejecución. Pasado ese tiempo, se le retira el control “por la fuerza”.

El “time slice” completo es un tiempo máximo, pero si durante ese tiempo la tarea pide algún recurso que no está, se la detiene antes a la espera del mismo. Luego se le transfiere el control a otra. También se la puede detener si otra tarea de mayor prioridad que estaba detenida pasa a estado “runnable”.

Si una tarea desea esperar, no debería hacer un loop, dado que visto desde el SO sigue trabajando. Lo que debe hacer es llamar a la función Sleep() que detiene su ejecución por la cantidad de milisegundos especificados.

Fin

¿Preguntas?