Planificación de uso de CPU

El sistema operativo debe *seleccionar* o *planificar* a qué proceso *RUNNABLE* se le asignará la cpu en diferentes momentos. Dependiendo de qué tipo de sistema y aplicaciones se ejecutan, ésta selección deberá hacerse en base a diferentes criterios.

En los *sistemas por lotes (batch systems)*, los programas se envían y se almacenan en un *jobs spool* para su ejecución y no necesariamente se ejecutan inmediatamente.

Un ***long term scheduler*** selecciona un proceso desde el *jobs spool* y lo ejecuta. En sistemas como *clusters de computadoras* o sistemas multiprocesadores de alto desempeño es común usar un *long term scheduler*. En estos sistemas los usuarios *submit jobs*. Un *job* consiste de al menos un programa mas información sobre los recursos requeridos, como número de cpus y memoria y posiblemente tiempo estimado de ejecución. Estos schedulers planifican el mejor plan de ejecución generalmente con el objetivo de *minimizar el tiempo medio de espera*.

Algunos algoritmos utilizados para lograr ese objetivo es el de ordenar los trabajos por menor tiempo de ejecución: *Shortest job first*.

En un sistema *time sharing* (como muchos SO modernos) no usan *long term schedulers*, permitiendo que un proceso comience su ejecución luego de su creación y luego usan ***short term schedulers*** que seleccionan un proceso en base a alguna política o criterio.

Existen diferentes criterios para la planificación del uso de cpu por parte de los procesos:

- ***Maximizar la utilización de la CPU***
- ***Rendimiento (throughput)***: Número de procesos completados por unidad de tiempo.
- ***Tiempo de ejecución de un proceso (turnaround)***
- ***Minimizar tiempos de espera por la cpu***
- ***Minimizar tiempos de respuesta***: Tiempo de procesamiento de un requerimiento.

Obviamente varios de estos criterios se contraponen entre sí. Por ejemplo, maximizar el rendimiento puede afectar negativamente a lograr bajos tiempos de respuesta.

Algoritmos de planificación

En ésta sección se tienen en cuenta los procesos en estado *READY o RUNNABLE*, es decir aquellos que están esperando por usar la cpu.

Fist-Come, Fist-Served (FCFS)

Este es el algoritmo más simple. Su implementa comúnmente con una cola (FIFO). Cuando un proceso se torna *RUNNABLE* se agrega al final de la cola y se le otorgará la cpu cuando esté primero. Su desventaja principal es que comúnmente produce tiempos promedio de espera grandes.

Este algoritmo no es *interrumpible (preemtive)*, es decir que hasta que el proceso no termina, no libera la CPU. Esto hace prácticamente imposible su uso en *time-sharing systems* en los que la cpu se *comparte* por intervalos.

Una variación consiste en que los procesos realicen *cooperative multitasking*, liberando la CPU voluntariamente mediante algún *syscall*.

Round-Robin (RR)

Esta técnica se implementa generalmente en sistemas *time sharing*. Es similar a FCFS pero con *preemption*. Al asignarse la CPU a un proceso se le asigna un ***quantum***.

> **Quantum**
>
> Intervalo máximo de tiempo de uso (*ráfaga*) de CPU. Generalmente se establece en una cantidad de ***ticks***. Un *tick* es un evento que ocurre en cada interrupción del *timer*, comúnmente cada algunos *milisegundos*.

Se implementa con una cola FIFO, donde se agrega un proceso *READY* al final y se selecciona el primero.

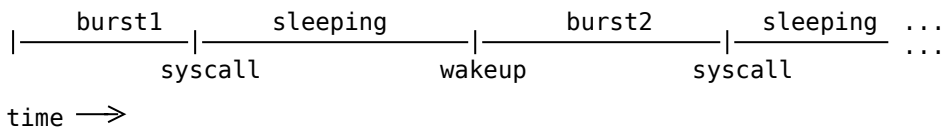
Si el proceso usó la cpu todo el *quantum*, es decir no hizo ninguna llamada al sistema que permita que el kernel tome el control, será interrumpido por el *timer*. En este caso se le quita la cpu y se lo encola nuevamente como `RUNNABLE` .

> **eos-steps**
>
> Ver el paso *07-preemtive* del proyecto *eos-steps*.

Shortest-Job first

Este algoritmo, comúnmente usado en *long term schedulers* o en *batch systems*. En estos casos el programador debe incluir en la descripción del *job* la duración estimada de ejecución (además de los requisitos de los recursos que requiere).

Esta idea también se puede aplicar en ***short-term schedulers*** asumiendo como tiempo de ejecución la duración de la próxima *ráfaga de cpu*. En éte caso, el algoritmo debería llamarse *shortest cpu-burst first*.

> **Ráfaga de cpu**
>
> Una ráfaga (*burst*) de cpu es el tiempo que un proceso usó la CPU desde que fue planificado hasta que libera la CPU.
>
> 
>

Se elige el proceso con menor tiempo de la próxima ráfaga. El principal problema es que la duración de la próxima rafaga no se conoce.

Una forma de **estimar** el tiempo de la próxima *ráfaga (burst)* es tener en cuenta su comportamiento pasado. Es común que un proceso repoduzca su comportamiento al menos temporalmente. Comúnmente se usa una función de ***promedio exponencial***. Sea t_n el tiempo de la última ráfaga y τ_n la última información. La duración de la próxima ráfaga se puede estimar como:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

El parámetro $0 \leq \alpha \leq 1$ controla el *peso relativo* al valor reciente y la *historia pasada*. Si $\alpha = 0$, entonces $\tau_{n+1} = \tau_n$ y la historia reciente (t_n) no se tiene en cuenta. Si $\alpha = 1$ sólo se considera la historia reciente. Un uso común es $\alpha = 0.5$.

Uso de prioridades

Es posible asignar *prioridades* a los procesos y elegir un con mayor prioridad. Generalmente se asignan números y su relación de orden puede ser arbitrario (a menor valor corresponde mayor prioridad o viceversa).

La asignación de prioridades puede ser estática (*at process creation time*) o dinámica (en *run time*).

Este algoritmo puede ser *preemtive* o no. Uno de los mayores problemas con este algoritmo es ***starvation***: La posibilidad que un proceso quede potencialmente sin planificar, por lo que se considera un algoritmo *no justo (unfair)*.

Una solución a este problema es aplicar ***envejecimiento (aging)***: *Incrementar gradualmente la prioridad de procesos que han estado esperando por un tiempo considerable*.

Multilevel queues

En este esquema, los procesos se clasifican en *grupos*. A cada grupo se le asigna una *prioridad*, o sea que es un algoritmo basado en prioridades.

Los procesos pueden *calificar* en un grupo debido a su comportamiento y se clasifican dinámicamente. Por ejemplo, un sistema podría considerar dos grupos de procesos:

1. *Interactivos*
2. En *background* (no interactivos) u *orientados a uso de CPU*.

donde generalmente se les da prioridad a los interactivos con el objetivo de minimizar el tiempo de respuesta ante los usuarios interactuando con el sistema.

Una implementación común usa *RR* en cada nivel (grupo) aunque se podrían usar diferentes algoritmos en cada nivel. El *scheduler* selecciona el primer proceso del grupo con mayor prioridad, como se muestra en la siguiente figura.

Cada nivel podría asociar un *quantum* diferente.

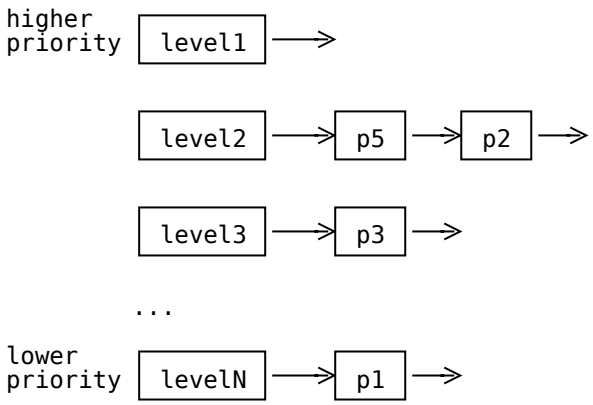


Figura 1: *Multilevel queue.*

La variante *multilevel feedback (mlf)* permite que los procesos se *muevan* o cambien su calificación dinámicamente. Por ejemplo, un proceso que no usó todo el *quantum* se considera *interactivo* y se sube de nivel, mientras que el que lo consumió completamente se considera en *background* u *orientado a cpu* y se lo baja. Esta estrategia prioriza a los procesos orientados a entrada-salida y contribuye a mejorar los tiempos de respuesta.

Planificación de threads

Algunos SO usan *kernel threads* para realizar *tareas diferidas* que generalmente se ejecutan fuera del contexto de una interrupción, como por ejemplo el *migration kernel thread* en Linux que se encarga de *medir* la carga de cpus y *migrar* los procesos a otras cpus para balancear la carga.

Otros *kernel threads* pueden usarse para procesar paquetes de red recibidos u otras tareas de larga duración, que no convienen realizar en el contexto de una interrupción.

Comúnmente los *kernel threads* tienen mayor prioridad que los procesos de usuario.

Caso de estudio: Linux scheduler

Los procesos y threads se representan en `struct task_struct`. Un proceso *multi-threaded* se modela con varias instancias de `task_struct`, donde una representa el hilo principal del proceso y los demás como hilos *hijos*. En este caso todos comparten el mismo *pid*.

Linux usa un scheduler basado en prioridades en el rango de 0 a 139. La prioridad de un proceso de usuario se determina en base a su valor *nice* (un atributo de un proceso) en el rango -20 a +19.

c

```
priority = nice + 20
```

La prioridad de un proceso lo categoriza en dos grandes clases:

- ***Real-time***: Prioridades 0-99. Se usan políticas *FIFO* (non-preemtive) o *RR* con *quantum* fijo.
- ***Normal***: Prioridades 100-140. Se usan políticas *RR* (con *quantum* variable), *BATCH* (para procesos no interactivos) e *IDLE* (los de menor prioridad).

Varios procesos del sistema y *kernel threads* ejecutan con prioridades de *real-time*.

> **Tiempo real**
>
> Un proceso o tarea de tiempo real requiere que el sistema le asigne los recursos (CPU, memoria, etc) en tiempos acotados (vencimientos). Ejemplo: Una aplicación que debe grabar un DVD requiere mantener el buffer de datos leídos de la fuente de origen con una cantidad mínima permanentemente ya que la grabación en un medio óptico es un proceso *continuo* que no se puede interrumpir (el láser no puede parar de grabar en el medio).

Los procesos normales inician con prioridad 0.

Las últimas versiones incluyen el *completely fair scheduler (CFS)* el cual usa un *red-black-tree* para ordenar los procesos por *tiempo de uso* en base a una *línea de tiempo* calculada. De esta forma el scheduler tiene costo constante por lo que se lo conoce como el $O(1)$ scheduler.

La llamada al sistema `nice(value)` permite asignar el valor de *nice*. Los usuarios comunes pueden invocar a *nice* con valores 0-19. Sólo el usuario *root* puede usar *nice* con valores entre -20 y 19.

Multiprocesadores

Algunas arquitecturas de hardware basados en multiprocesadores se desarrollaron según el concepto de *asymmetric multiprocessing (amp)*. En estas arquitecturas una CPU (*master*) generalmente ejecuta el código del kernel y sus servicios mientras que los procesos de usuarios se ejecutan en las demás cpus (*workers*). El scheduler corre en el *master*, por lo cual simplifica los problemas de paralelismo y concurrencia.

Las arquitecturas modernas son multi-procesadores simétricos (*smp*), es decir que todas las cpus ejecutan código de usuario y del kernel.

En una arquitectura *smp* se debe poner especial cuidado en los problemas de concurrencia generados por el paralelismo ya que varias cpus pueden estar ejecutando el mismo código del cliente en paralelo.

En un multiprocesador uno de los desafíos es lograr un buen *balance de carga (load balancing)*: Emparejar la carga de trabajo entre todas las cpus.

Esto require que el sistema pueda contabilizar la carga y asignar a las cpus menos cargadas. Así en cada *context switch* es posible que un proceso *migre de cpu*, es decir, que continúe su ejecución en otra cpu.

La *migración* puede afectar al rendimiento, ya que puede que la memoria *caché* de la cpu anterior debe quedar invalidada y la caché de la nueva cpu debe ir *llenándose* al continuar la ejecución del proceso.

Es posible ver que estos dos objetivos se contraponen, por lo que hay que lograr una solución de compromiso, por lo cual muchos schedulers implementan *processor affinity*, que *tratan* de minimizar la migración. En algunos sistemas, como Linux, proveen llamadas al sistema para pedirle al sistema que no migre de cpu al proceso (*hard affinity*). Para más detalles, ver **set_affinity syscall**.

Una arquitectura *multi-core* replica varias cpus en un mismo chip. A suvez cada *core* puede usar múltiples *pipelines* para permitir ejecutar varios *threads* o *hilos de ejecución* en paralelo para minimizar los *memory stalls* de un core. Una *contención* se produce cuando un core tiene que *esperar* por datos en memoria que aún no se han cargado (*prefetch*) en la caché (*cache miss*).

> **Hyperthreading**

Capacidad de que un *core* pueda ejecutar más de un thread en paralelo. Comúnmente se implementa asignando varios *pipelines* al core. Así, una CPU de dos cores con dos pipelines cada uno puede verse como un multiprocesador con 4 CPUs.

In RISC-V, un *hart* es una abstracción de un *hardware thread* o *logical processor*: Una unidad lógica de procesamiento con su propio conjunto de registros. Una implementación puede hacerlo mediante *cores* que usan múltiples pipelines.

Bibliografía

1. G. Wolf, E. Ruiz, F. Bergero, E. Meza. **Fundamentos de Sistemas Operativos**. 2015. Sección 4.2.

2. A. Tanenbaum, H. Bos. *Modern Operating Systems*. Section 2.4.

3. A. Tanenbaum, H. Bos. *Modern Operating Systems*. Part II, Chapter 5: CPU scheduling.

Anterior
Procesos y threads

Próximo
Concurrencia e IPC