

Scheduling

Rodolfo Baader¹

¹Departamento de Computación, FCEyN,
Universidad de Buenos Aires, Buenos Aires, Argentina

Sistemas Operativos, segundo cuatrimestre de 2024


(2) Estado de situación...

- En la clase pasada vimos:
 - IPC.
- Hoy:
 - Vamos a poner la lupa en el scheduler.


(3) Scheduling

- La política de scheduling es una de las principales huellas de identidad de un SO.
- Es tan importante que algunos SO proveen más de una.
- Buena parte del esfuerzo por optimizar el rendimiento de un SO se gasta en la política de scheduling.


(4) Objetivos de la política de scheduling

- Qué optimizar: 
 - Ecuanimidad (*fairness*): que cada proceso reciba una dosis “justa” de CPU (para alguna definición de justicia).
 - Eficiencia: tratar de que la CPU esté ocupada todo el tiempo.
 - Carga del sistema: minimizar la cant. de procesos listos que están esperando CPU.
 - Tiempo de respuesta: minimizar el tiempo de respuesta *percibido* por los usuarios interactivos.
 - Latencia: minimizar el tiempo requerido para que un proceso empiece a dar resultados.
 - Tiempo de ejecución: minimizar el tiempo total que le toma a un proceso ejecutar completamente.
 - Rendimiento (*throughput*): maximizar el número de procesos terminados por unidad de tiempo.
 - Liberación de recursos: hacer que terminen cuanto antes los procesos que tiene reservados más recursos.
- Muchos de estos objetivos son contradictorios.

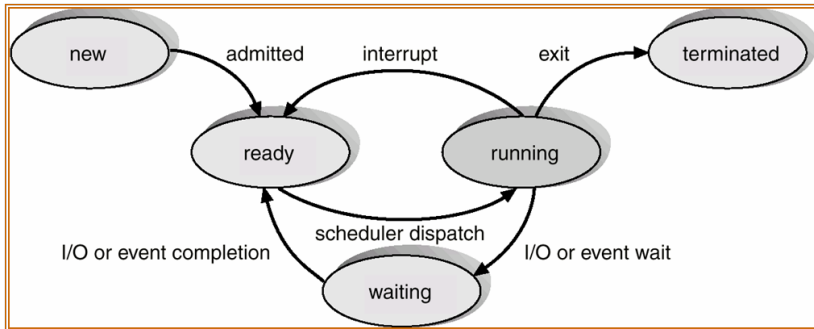
(5) Objetivos de la política de scheduling (cont.)

- Si los usuarios del sistema son heterogéneos, pueden tener distintos intereses.
- Una cosa queda clara: no se puede tener ICI20yIMdHC.
- En definitiva, cada política de scheduling va a buscar maximizar una función objetivo, que va a ser una combinación de estas metas tratando de impactar lo menos posible en el resto. 

(6) Cuando actúa el scheduler

- El scheduling puede ser *cooperativo* o *con desalojo*. 
- Si es con desalojo (también llamado scheduling apropiativo o *preemptive*), el scheduler se vale de la interrupción del clock para decidir si el proceso actual debe seguir ejecutándose o le toca a otro.
- Recordemos: el clock interrumpe 50 o 60 veces/seg.
- Si bien suele ser deseable, el scheduling con desalojo:
 - Requiere un clock con interrupciones (podría no estar disponible en procesadores embebidos).
 - No le da garantías de continuidad a los procesos (podría ser un problema en SO de tiempo real).
- Cuando tenemos multitarea cooperativa,
 - El scheduler analiza la situación cuando el kernel toma control (en los syscalls).
 - Especialmente cuando el proceso hace E/S.
 - A veces se proveen llamadas explícitas para permitir que se ejecuten otros procesos.
- En realidad, los schedulers con desalojo combinan ambos enfoques.

(7) Proceso: estado



(8) El procesador como una sala de espera

- Un enfoque posible es FIFO, también conocido como *FCFS* (*First Came, First Served*).
- El problema es que supone que todos los procesos son iguales.
- Si llega un “megaproceso” que requiere mucha CPU, tapona a todos los demás.
- Entonces, agreguémosle prioridades al modelo. Como en una sala de espera.
- Posible problema: *inanición* (*starvation*). Los procesos de mayor prioridad demoran infinitamente a los de menor prioridad, que nunca se ejecutan. ⚠
- Una posible solución: aumentar la prioridad de los procesos a medida que van “envejeciendo”.
- Cualquier esquema de prioridades fijas corre riesgo de inanición. ⚠

(9) Round robin

- La idea es darle un quantum a cada proceso, e ir alternando entre ellos.
- ¿Cuánto dura el quantum?
 - Si es muy largo, en SO interactivos podría parecer que el sistema no responde.
 - Si es muy corto, el tiempo de scheduling+context switch se vuelve una proporción importante del quantum. Por ende, el sistema pasa un porcentaje alto de su tiempo haciendo “mantenimiento” en lugar de trabajo de verdad.
- Se lo suele combinar con prioridades.
 - Que pueden estar dadas por el tipo de usuario (administrativas) o pueden ser “decididas” por el propio proceso. Esto último no suele funcionar.
 - Que van decreciendo a medida que los procesos reciben su quantum, para evitar inanición de los otros.
- Además, los procesos que hacen E/S suelen recibir crédito extra, por ser buenos compañeros.

(10) Múltiples colas

- “Otra alternativa es tener múltiples colas”, dijo el perro.
- (Tarea: Un perro con múltiples colas, ¿es más feliz?)
- Colas con 1, 2, 4, 8 quanta c/u.
- A la hora de elegir un proceso la prioridad la tiene siempre la cola con menos quanta.
- Cuando a un proceso no le alcanza su cuota de CPU es pasado a la cola siguiente, lo que disminuye su prioridad, pero le asigna más tiempo de CPU en el próximo turno.
- Los procesos de máxima prioridad, los interactivos en gral, van a la cola de máxima prioridad.
- Se puede hacer que cuando un proceso termina de hacer E/S vuelva a la cola de máxima prioridad, porque se supone que va a volver a hacerse interactivo.
- La idea general es minimizar el tiempo de respuesta para los procesos interactivos, suponiendo que los cálculos largos son menos sensibles a demoras.

(11) Trabajo más corto primero

- También llamada *SJF* (*Shortest Job First*).
- Está ideada para sistemas donde predominan los trabajos batch. Está orientada a maximizar el throughput.
- En esos casos, muchas veces se puede predecir la duración del trabajo o al menos clasificarlo (por ejemplo: menos de 10', menos de 30', menos de 60', más de 60').
- Si conozco las duraciones de antemano, es óptimo (en cuanto a la latencia promedio).
- Otra alternativa es no pensar en la duración total, sino más bien en cuánto tiempo necesita hasta hacer E/S de nuevo.
- El problema real es cómo saber cuánta CPU va a necesitar un proceso.
- Una alternativa es usar la info del pasado para predecir.
- Puede salir mal si los procesos tienen comportamiento irregular.

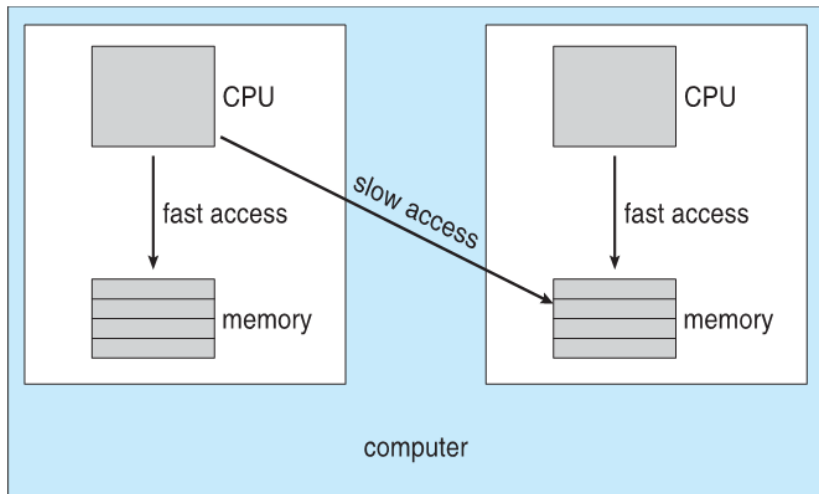
(12) Scheduling para RT

- Los sistemas de tiempo real son aquellos en donde las tareas tiene fechas de finalización (*deadlines*) estrictas.
- En general se usan en entornos críticos: si un deadline no se cumple, algo malo pasa.
- Scheduling en RT es un problema en sí mismo. Apenas vamos a mencionarlo.
- Una política posible consiste en correr el proceso más cercano a perder su deadline (EDF: *Earliest-Deadline-First*).

(13) Scheduling en SMP

- Scheduling en SMP es también un problema bastante distinto.
- El problema es el caché, que es de vital importancia para el rendimiento de los programas.
- Si la política de scheduling hace pasar un proceso a otro procesador, éste llega con el caché vacío, tardando mucho más de lo que tardaría si se hubiese ejecutado en el mismo procesador que antes.
- Por eso se utiliza el concepto de *afinidad al procesador*: tratar de usar el mismo procesador, aunque se tarde un poco más en obtenerlo.
- Si esto se respeta a rajatabla, *afinidad dura*. Si simplemente es un intento, *afinidad blanda*.
- A veces se intenta distribuir la carga entre todos los procesadores:
 - Push migration.
 - Pull migration.

(14) NUMA



(15) En la práctica...

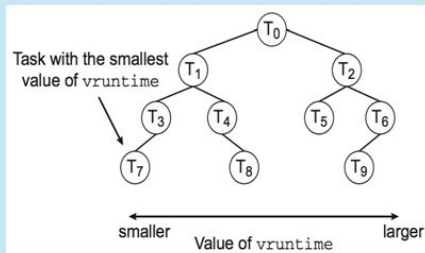
- Muchas de las consideraciones que planteamos tienen a su vez, bemoles. Por ejemplo:
 - ¿El scheduling debe ser justo entre procesos o usuarios? Un usuario puede tener varios procesos...
 - ¿Qué pasa con los procesos hijos?
 - Si un proceso requiere mucha CPU, ¿debo priorizarlo o matarlo? Tal vez tenga usuarios hostiles que estén tratando de abusar el sistema...
- Elegir un buen algoritmo de scheduling que funcione en la práctica es muy difícil.
- Suele requerir prueba/error/corrección, y muchas veces deben ajustarse a medida que cambian los patrones de uso.
- A veces se arman modelos matemáticos basados en teoría de colas.
- Otras, se prueban con patrones de carga tomados de sistemas concretos o benchmarks estandarizados.

(16) En la práctica... (cont.)

- Si bien cada proceso es único, algunas cosas se pueden saber:
 - Si un proceso abre una terminal, muy probablemente esté por convertirse en interactivo.
 - En algunos casos se puede usar análisis estático para ver si cierto comportamiento se va a repetir, si el proceso no tiene pensado terminar, etc.
 - Etc.
- Al cóctel le faltan aún ingredientes...
 - Usos específicos: ejemplo: motores de BD, cómputo científico, micro-/macro- benchmarking.
 - Threads (por ahora pueden pensarlo como procesos dentro de procesos).
 - Virtualización.

(17) Linux CFS Scheduler

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(\lg N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

(18) Tarea

- Leer los detalles del libro.
- En especial, seguir los ejemplos numéricos para entender los algoritmos.

(19) Dónde estamos

- Vimos
 - Todas las consideraciones, contradictorias a veces, que deben hacerse a la hora de elegir un algoritmo de scheduling.
 - Analizamos los algoritmos más comunes.
- Próxima teórica:
 - Sincronización entre procesos.