16 CPU Scheduling

Readings for this topic: Silberschatz/Galvin/Gagne Chapter 5

Until now you have heard about processes and memory. From now on you'll hear about *resources*, the things operated upon by processes. Resources range from cpu time to disk space to channel I/O time.

Resources fall into two classes:

- Preemptible: processor or I/O channel. Can take resource away, use it for something else, then give it back later.
- Non-preemptible: once given, it can't be reused until process gives it back. Examples are file space, terminal, and maybe memory.

This distinction is a little arbitrary, since anything is preemptible if it can be saved and restored. It generally measures the difficulty of preemption.

OS makes two related kinds of decisions about resources:

- Allocation: who gets what. Given a set of requests for resources, which processes should be given which resources in order to make most efficient use of the resources? Implication is that resources aren't easily preemptible.
- Scheduling: how long can they keep it. When more resources are requested than can be granted immediately, in which order should they be serviced? Examples are processor scheduling (one processor, many processes), memory scheduling in virtual memory systems. Implication is that resource is preemptible.

Resource #1: the processor.

Processes may be in any one of three general scheduling states:

- Running.
- Ready. That is, waiting for CPU time. Scheduler and dispatcher determine transitions between this and running state.

- Blocked. Waiting for some other event: disk I/O, message, semaphore, etc.

Goals for scheduling disciplines:

- Efficiency of resource utilization. (keep CPU and disks busy)
- Minimize overhead. (context switches)
- Minimize response time.
- Distribute cycles equitably. What does this mean?

FIFO (also called FCFS): run until finished.

- In the simplest case this means uniprogramming.
- Usually, "finished" means "blocked". One process can use CPU while another waits on a semaphore. Go to back of run queue when ready.
- Problem: one process can monopolize CPU.

Solution: limit maximum amount of time that a process can run without a context switch. This time is called a *time slice*.

Round Robin: run process for one time slice, then move to back of queue. Each process gets equal share of the CPU. Most systems use some variant of this. What happens if the time slice isn't chosen carefully?

- Too long: one process can monopolize the CPU.
- Too small: too much time wasted in context swaps.

Originally, Unix had 1 sec. time slices. Too long. Most systems today use time slices of 10 - 100 milliseconds.

Implementation of priorities: run highest-priority processes first, use round-robin among processes of equal priority. Re-insert process in run queue behind all processes of greater or equal priority.

Round-robin can produce poor response time. Go through example of ten processes each requiring 100 time slices. They each take about 1000 time slices to finish, whereas in FIFO they would average 550 time slices to finish. This is like government: it is (supposedly) fair, but uniformly inefficient.

Another example: User types text using emacs; emacs process competes with 10 compute-bound processes. With a timeslice of 100ms, each key stroke has a response time of $10 * 100ms = 1s$!.

What is the best we can do wrt response time? STCF: shortest time to completion first with preemption. This minimizes the average response time.

As an example, show two processes, one doing 1 ms computation followed by 10 ms I/O, one doing all computation. Suppose we use 100 ms time slice: I/O process only runs at 1/10th speed, I/O devices are only utilized 10% of time. Suppose we use 1 ms time slice: then compute-bound process gets interrupted 9 times unnecessarily for each valid interrupt. STCF works quite nicely.

Unfortunately, STCF requires knowledge of the future. Instead, we can use *past* performance to predict *future* performance.

If a process has already taken a long time, it will probably take a long time more. We use priorities to disfavor long processes.

Exponential Queue (or Multi-Level Feedback Queues): attacks both efficiency and response time problems.

- Give newly runnable process a high priority and a very short time slice. If process uses up the time slice without blocking then decrease priority by 1 and double time slice for next time.
- Go through the above example, where the initial values are 1ms and priority 100.
- Techniques like this one are called *adaptive*. They are common in interactive systems.

- The CTSS system (MIT, early 1960's) was the first to use exponential queues.

Fair-share scheduling (similar to what's implemented in Unix):

- Keep history of recent CPU usage for each process.
- Give highest priority to process that has used the least CPU time recently. Highly interactive jobs, like emacs, will use little CPU time and get high priority. CPU-bound jobs, like compiles, will get lower priority.
- Can adjust priorities by changing "billing factors" for processes. E.g. to make high-priority process, only use half its recent CPU usage in computing history.

Example: 4.4 BSD UNIX scheduler

Priorities in the range [0,127] (0 highest, 0-49 reserved for kernel)

In PCB:
p_usrpri: user-mode scheduling priority of a process
p_estcpu: estimate of recent CPU utilization of a process
p_nice: user-settable weighing factor in range [-20,20]

User priority recalculated every 4 clock ticks (40ms) as follows:
$p\_usrpri \leftarrow 50 + p\_estcpu/4 + 2 * p\_nice$

Resulting value is capped into range [50,127]

p_estcpu is incremented every time the clock ticks (10ms) and the process is executing. In addition, once per second, CPU utilization of a runnable process is adjusted as:

$p\_estcpu \leftarrow (2 * load/(2 * load + 1)) * p\_estcpu + p\_nice$

load: sampled average of the sum of lengths of run queue and short-term sleep queue (e.g., processes waiting for disk I/O).

Example: let T_i be the number of clock ticks accumulated during interval i. Assume $load = 1$.

$p\_estcpu \leftarrow 0.66 * T_0$
$p\_estcpu \leftarrow 0.66 * (T_1 + 0.66 * T_0) = 0.66 * T_1 + 0.44 * T_0$
$p\_estcpu \leftarrow 0.66 * T_2 + 0.44 * T_1 + 0.30 * T_0$
$p\_estcpu \leftarrow 0.66 * T_3 + ... + 0.20 * T_0$
$p\_estcpu \leftarrow 0.66 * T_4 + ... + 0.13 * T_0$

$\Rightarrow$ about 90% of CPU utilization forgotten after 5s.

For sleeping processes, p_estcpu is adjusted when the process wakes us:
$p\_estcpu \leftarrow (2 * load/(2 * load + 1))^{p\_slptime} * p\_estcpu$

$p\_slptime$: time process spent sleeping in seconds

Lottery scheduling:

- different approach to scheduling/dispatch (not based on priorities)
- every runnable process owns a set of "lottery tickets" from a fixed sized ticket pool (range of number from 0-N)
- fraction of ticket pool allocated to a process is directly proportional to target CPU share
- during dispatch, OS draws a uniformly distributed random number in the range 0-N
- process that owns the matching ticket gets the CPU

Start-time fair queuing (STFQ):

- maintain virtual time T_i for each process i, initially $T_i \leftarrow realTime$
- after process has executed for time t, advance $T_i \leftarrow T_i + (1/r_i) * t$, where r_i is process i's desired CPU share among the currently runnable processes.
- always schedule runnable process with minimal T_i

- what happens if a process is not runnable for a while? T_i will fall behind, giving i a potentially huge advantage when it wakes up. Solution options:
  - upon wake-up, set $T_i \leftarrow realTime$
  - upon wake-up, set $T_i \leftarrow T_i + sleepTime$
  - upon wake-up, set $T_i \leftarrow T_i + sleepTime - d$, where d is a system parameter reflecting the amount of CPU time a sleeping process may "save up".

Earliest deadline first:

- real-time scheduling algorithm
- requires a priori knowledge of deadline and CPU requirements

Rate-based scheduling:

- grant a certain amount of CPU time at a fixed period
- Example: 1ms of CPU time every 100ms
- in general, requires a priori knowledge of CPU requirements

Summary:

- In principle, scheduling algorithms can be arbitrary, since the system should produce the same results in any event.
- However, the algorithms have strong effects on the system's overhead, efficiency, and response time.
- The best schemes are adaptive. To do absolutely best, we'd have to be able to predict the future.

Best scheduling algorithms tend to give highest priority to the processes that need the least! Reagonomics?