CPU Scheduling

The CPU scheduler makes a sequence of “moves” that determines the interleaving of threads.
- Programs use synchronization to prevent “bad moves”.
- ... but otherwise scheduling choices appear (to the program) to be non-deterministic.

The scheduler’s moves are dictated by a scheduling policy.

Scheduler's moves are dictated by a scheduling policy.

Scheduler Goals

- **response time** or latency
  - How long does it take to do what I asked? (R)
- **throughput**
  - How many operations complete per unit of time? (X)
  - Utilization: what percentage of time does the CPU (and each device) spend doing useful work? (U)
  - “Keep things running smoothly.”
- **fairness**
  - What does this mean? Divide the pie evenly? Guarantee low variance in response times? Freedom from starvation?
- **meet deadlines and guarantee jitter-free periodic tasks**

Outline

1. **the CPU scheduling problem, and goals of the scheduler**
   - Consider preemptive timeslicing.
2. **fundamental scheduling disciplines**
   - FCFS: first-come-first-served
   - SJF: shortest-job-first
3. **practical CPU scheduling**
   - Multilevel feedback queues: using internal priority to create a hybrid of FIFO and SJF.
   - Proportional share

A Simple Policy: FCFS

The most basic scheduling policy is **first-come-first-served**, also called **first-in-first-out** (FIFO).
- FCFS is just like the checkout line at the QuickiMart.
  - Maintain a queue ordered by time of arrival.
  - GetNextToRun selects from the front of the queue.
- FCFS with preemptive timeslicing is called **round robin**.
  - Preemption quantum (timeslice): 5-800 ms.

Evaluating FCFS

How well does FCFS achieve the goals of a scheduler?
- **throughput**. FCFS is as good as any non-preemptive policy.
  - ... if the CPU is the only schedulable resource in the system.
- **fairness**. FCFS is intuitively fair... sort of.
  - “The early bird gets the worm”... and everyone else is fed eventually.
- **response time**. Long jobs keep everyone else waiting.
Preemptive FCFS: Round Robin

Preemptive timeslicing is one way to improve fairness of FCFS. If a job does not block or exit, force an involuntary context switch after each quantum $Q$ of CPU time (its timeslice in Linux lingo). Preempted job goes back to the tail of the ready list. With infinitesimal $Q$, round robin is called processor sharing.

$D_1 = 3$ $D_2 = 1$ $R = (3 + 3 + 6 + 3\varepsilon)/3 = 4 + \varepsilon$

In this case, $R$ is unchanged by timeslicing. Is this always true?

Evaluating Round Robin

- **Response time.** RR reduces response time for short jobs. For a given load, a job’s wait time is proportional to its $D$.
- **Fairness.** RR reduces variance in wait times.
- **Throughput.** RR imposes extra context switch overhead. CPU is only $Q/(Q+\varepsilon)$ as fast as it was before. Degrades to FCFS-RTC with large $Q$.

$
\begin{align*}
D_1 &= 5 \\
D_2 &= 1 \\
R &= (5+6+\varepsilon)/2 = 4 + \varepsilon
\end{align*}
$

Digression: RR and System Throughput II

On a multiprocessor, RR may improve throughput under light load:

- **The scenario:** three salmon steaks must cook for 5 minutes per side, but there’s only room for two steaks on the hibachi. 30 minutes worth of grill time needed: steaks 1, 2, 3 with sides A and B.
- **FCFS-RTC:** steaks 1 and 2 for 10 minutes, steak 3 for 10 minutes. Completes in 20 minutes with grill utilization a measly 75%.
- **RR:** steaks 1A and 2A...flip...1B and 3A...flip...2B and 3B. Completes in three quanta (15 minutes) with 100% utilization.
- **RR** may speed up parallel programs if their inherent parallelism is poorly matched to the real parallelism. E.g., 17 threads execute for 3 time units on 16 processors.

Minimizing Response Time: SJF

**Shortest Job First (SJF)** is provably optimal if the goal is to **minimize $R$.**

**Example:** express lanes at the MegaMart

**Idea:** get short jobs out of the way quickly to minimize the number of jobs waiting while a long job runs. **Intuition:** longest jobs do the least possible damage to the wait times of their competitors.

$D_1 = 1$ $D_2 = 2$ $D_3 = 3$ $R = (1 + 3 + 6\varepsilon)/3 = 3 + \varepsilon$

Behavior of SJF Scheduling

Little’s Law does not hold if the scheduler considers a priori knowledge of service demands, as in SJF.

- **With SJF**, best-case $R$ is not affected by the number of tasks in the system. Shortest jobs budge to the front of the line.
- **Worst-case $R$** is unbounded, just like FCFS. Since the queue is not “fair”, we call this starvation: the longest jobs are repeatedly denied the CPU resource while other more recent jobs continue to be fed.
- **SJF** sacrifices fairness to lower average response time.
- **Contestitively**, SJF (or Shortest Remaining Processing Time) may be a very good policy in practice, if there is a small number of very long jobs (e.g., the Web).

SJF in Practice

Pure SJF is impractical: scheduler cannot predict $D$ values. However, SJF has value in real systems:

- Many applications execute a sequence of short CPU bursts with I/O in between.
- E.g., interactive jobs block repeatedly to accept user input. Goal: deliver the best response time to the user.
- E.g., jobs may go through periods of I/O-intensive activity. Goal: request next I/O operation ASAP to keep devices busy and deliver the best overall throughput.
- Use adaptive internal priority to incorporate SJF into RR. Weather report strategy: predict future $D$ from the recent past.
Priority

Some goals can be met by incorporating a notion of priority into a “base” scheduling discipline.

Each job in the ready pool has an associated priority value; the scheduler favors jobs with higher priority values.

External priority values:
- imposed on the system from outside
- reflect external preferences for particular users or tasks
  “All jobs are equal, but some jobs are more equal than others.”
- Example: Unix nice system call to lower priority of a task.
- Example: Urgent tasks in a real-time process control system.

Internal Priority

Internal priority: system adjusts priority values internally as an implementation technique within the scheduler.

- drop priority of jobs consuming more than their share
- boost jobs that already hold resources that are in demand
e.g., internal sleep primitive in Unix kernels
- boost jobs that have starved in the recent past
- typically a continuous, dynamic, readjustment in response to observed conditions and events
  may be visible and controllable to other parts of the system

Two Schedules for CPU/Disk

Naive Round Robin
25/25: U = 100%

Disk busy 15/25: U = 60%

100% performance improvement

Round Robin with SJF
25/37: U = 67%

Disk busy 15/37: U = 40%

33% performance improvement

Multilevel Feedback Queue

Many systems (e.g., Unix variants) implement priority and incorporate SJF by using a multilevel feedback queue.

- multilevel: Separate queue for each of N priority levels.
- feedback: Factor previous behavior into new job priority.

Note for CPS 196 Spring 2006

We did not discuss real-time scheduling or reservations and time constraints as in Microsoft’s Rialto project. The following Rialto slides are provided for interest only. The other slides I did not use, but they may be helpful for completeness.

Rialto

Real-time schedulers must support regular, periodic execution of tasks (e.g., continuous media).
Microsoft’s Rialto scheduler [Jones97] supports an external interface for:

- CPU Reservations
  “I need to execute for X out of every Y units.”
  Scheduler exercises admission control at reservation time: application must handle failure of a reservation request.
- Time Constraints
  “Run this before my deadline at time T.”
A Rialto Schedule

Rialto schedules constrained tasks according to a static task graph.

- For each base period, pick a path from root to a leaf.
- At each visited node, execute associated task for specified time $t$.
- Visit subsequent leaves in subsequent base periods.
- Modify the schedule only at request time.

Considering I/O

In real systems, overall system performance is determined by the interactions of multiple service centers.

A queue network has $K$ service centers. Each job makes $V_k$ visits to center $k$, demanding service $S_k$.

- Forced Flow Law: $U_k = \lambda_k S_k = \lambda_k D_k$ (Arrivals/throughputs $\lambda_k$ at different centers are proportional.)
- Easy to predict $X_k$, $U_k$, $\lambda_k$, and $R_k$ at each center; use Forced Flow Law to predict arrival rate $\lambda_k$ at each center.
- Then, apply Little's Law to $k$.

I/O and Bottlenecks

It is easy to see that the maximum throughput $X$ of a system is reached as $1/\lambda$ approaches $D_k$ for service center $k$ with the highest demand $D_k$.

$k$ is called the bottleneck center.

- Overall system throughput is limited by $\lambda_k$ when $U_k$ approaches 1.

Preemption

Scheduling policies may be preemptive or non-preemptive.

Preemptive: scheduler may unilaterally force a task to relinquish the processor before the task blocks, yields, or completes.

- timeslicing prevents jobs from monopolizing the CPU
- Scheduler chooses a job and runs it for a quantum of CPU time.
- A job executing longer than its quantum is forced to yield by scheduler code running from the clock interrupt handler.

- use preemption to honor priorities
- Preempt a job if a higher priority job enters the ready state.