Lecture 11: CPU Scheduling

11.0 Main Points

Scheduling policy goals
Policy options
Implementation considerations

Earlier, said dispatcher can choose any thread on the ready list to run. But how is OS to decide, when it has a choice?

11.1 Scheduling Policy Goals

1. **Minimize response time**: elapsed time to do an operation (or job)
   Response time is what the user sees: elapsed time to
   echo a keystroke in editor
   compile a program
   run a large scientific problem

2. **Maximize throughput**: operations (or jobs) per second
   (Throughput is related to response time, but they're not identical)

Two parts to maximizing throughput
   a. **Minimize overhead** (for example, context switching)
   b. **Efficient use of system resources** (not only CPU, but disk, memory, etc.)

3. **Fair**: share CPU among users in some equitable way

What does fairness mean?

Minimize average response time? We will argue fairness is a tradeoff against average response time; can get better average response time by making system **less** fair.

11.2 Assumptions

Bunch of algorithms for CPU scheduling -- big area of research in the early 70's.

These assume:
one program per user
one thread per program
programs are independent

Clearly, these are unrealistic. Simplify problem so it can be solved. Open issue is: what happens if you remove these constraints?

10.3 Scheduling policies:

10.3.1 FIFO

Different names for the same thing:
FCFS -- first come first serve
FIFO -- first in first out
Run until done

In early systems, FIFO meant, one program kept CPU until it completely finished. With strict uniprogramming, if have to wait for I/O, keep processor.

Later, FIFO just means, keep CPU until thread blocks (this is what I'll assume).

FIFO pros&cons:
+ simple
- short jobs get stuck behind long jobs

10.3.2 Round Robin

Solution? Add timer, and preempt CPU from long-running jobs. Just about every real operating system does something of this flavor.

Round-robin: after time slice, move thread to back of the queue
In some sense, it's fair -- each job gets equal shot at the CPU.

10.3.2.1 How do you choose time slice?

1) What if too big?
   Response time suffers

2) What if too small?
   Throughput suffers. Spend all your time context switching, none getting real work done.

In practice, need to balance these two. Typical time slice today is between 10-100 milliseconds; typical timeslice is 0.1 - 1 millisecond,
so roughly 1% overhead due to time-slicing.

10.3.2.2 Comparison between FIFO and Round Robin

Assuming zero-cost time slice, is RR always better than FIFO?
For example: 10 jobs, each take 100 seconds of CPU time.
Round Robin time slice of1 second. All start at the same time:

Job completion times

Job #	FIFO	Round Robin
1	100	991
2	200	992
...	...	...
9	900	999
10	1000	1000

Round robin runs one second from each job, before going back to the first. So each job accumulates 99 seconds of CPU time before any
finish.

Both round robin and FIFO finish at the same time, but **average** response time is much worse under RR than under FIFO.

Thus, round robin pros&cons:
+ better for short jobs
- poor when jobs are same length

10.3.3 STCF/SRTCF

STCF: shortest time to completion first. Run whatever job has the least amount of stuff to do.

SRTCF: shortest remaining time to completion first.
Preemptive version of STCF -- if job arrives that has a shorter time to completion than the remaining time on the current job, immediately preempt CPU to give to new job.

Idea is get short jobs out of the system. Big effect on short jobs, small effect on short jobs. Result is better **average** response time

In fact, STCF/SRTCF are the best you can possibly do, at minimizing average response time (STCF among non-preemptive policies, SRTCF among preemptive policies). Can prove they're optimal. Since SRTCF is always at least as good as STCF, focus on SRTCF.

10.3.3.1 Comparison of SRTCF with FIFO and Round Robin

What if all jobs are the same length? SRTCF becomes the same as FIFO (in other words, FIFO is as good as you can do if all jobs are the same length).

What if jobs have varying length? SRTCF (and round robin): short jobs don't get stuck behind long jobs.

Example to illustrate the benefits of SRTCF:

Three jobs:
A, B: both CPU bound, infinite length
C: I/O bound, loop
    1 ms of CPU
    10 ms of disk I/O

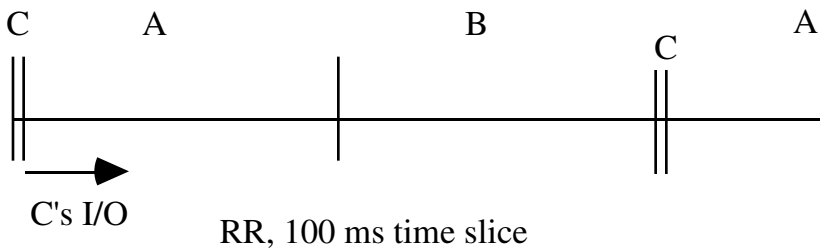
By itself, C uses 90% of the disk; by itself, A or B could use 100% of the CPU. What happens if we try to share system between A, B, and C?

With FIFO:
    once A or B get in, keep CPU forever.

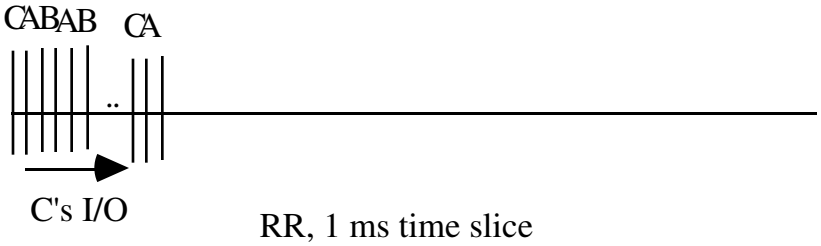
With Round Robin (100 ms time slice):
    only get 5% disk utilization

With Round Robin (1 ms time slice):
    Get nearly 90% disk utilization -- almost as good as C alone.
But we haven't slowed A or B by all that much: they still get 90% of the CPU. (Lots of wakeups, however!)

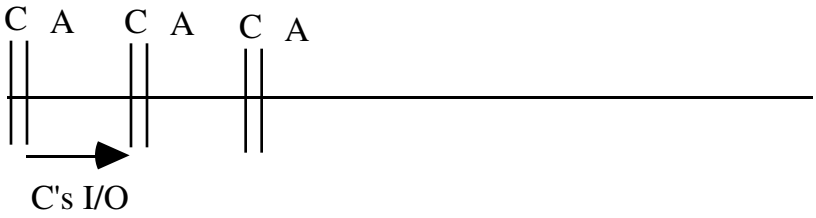
With SRTCF: no needless preemptions (run C as soon as possible, run either A or B to completion)



RR, 100 ms time slice



RR, 1 ms time slice



SRCTF

Effect of RR time quanta and SRTCF on I/O bound jobs

A downside to SRTCF is that it can lead to starvation. Lots of short jobs can keep long jobs from making any progress.

SRTCF pros&cons:

+ optimal (average response time)
- hard to predict the future
- unfair

10.3.3.2 Knowledge of future

Problem: STCF/SRTCF require knowledge of the future.

How do you know how long program will run for?

Some systems ask the user:when you submit a job like a compile, have to say how long it will take.

To stop cheating, if your job takes more than what you said, system kills your job. Start all over. Like with the Banker's algorithm -- hard to predict resource usage in advance.

Instead, can't really know how long things will take, but can use SRTCF as a yardstick, for measuring other policies. Optimal, so can't do any better than that!

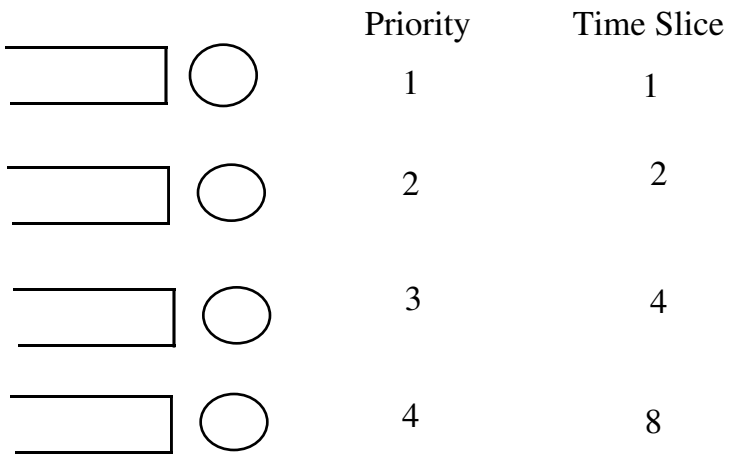
10.3.4 Multilevel feedback

Central idea in computer science (occurs in lots of places): **use past to predict future**. If program was I/O bound in past, likely to be in future.
If computer behavior was random, induction wouldn't help. Or if past behavior was opposite of current behavior.

But program behavior is regular, most of the time. How do we exploit this? If past behavior predicts future behavior, then favor jobs that have been at CPU least amount of time, to approximate SRTCF!

Adaptive policies: change policy based on past behavior. Used in CPU scheduling, in virtual memory, in file systems, etc.

Multi-level feedback queues (first used in CTSS, example of an adaptive policy for CPU scheduling): multiple queues, each with different priority. OS does round robin at each priority level -- run highest priority jobs first; once those finish, run next highest priority, etc. Round robin time slice increases exponentially at lower priorities.



Multilevel feedback queues

Adjust each job's priority as follows (details vary):

1. Job starts in highest priority queue.
2. If timeout expires, drop one level
3. If timeout doesn't expire, push up one level (or back to top)

Result approximates SRTCF: CPU bound jobs drop like a rock, while short-running I/O bound jobs stay near top.

Multilevel feedback queues (like SRTCF) are still unfair: long running jobs may never get the CPU.

Countermeasure: user action that can foil intent of the OS designer. For multilevel feedback, countermeasure would be to put in meaningless I/O to keep job's priority high. Of course, if everyone did this, wouldn't work!

10.3.5 Lottery scheduling

What should we do about fairness? Since SRTCF is optimal and unfair, any increase in fairness (for instance by giving long jobs a fraction of the CPU, even when there are shorter jobs to run) will have to hurt average response time.

How do we implement fairness?

Could give each queue a fraction of the CPU, but this isn't always fair. What if there's one long-running job, and 100 short-running ones?

Could adjust priorities: increase priority of jobs, as they **don't** get service. This is what's done in UNIX.

Problem is that this is ad hoc - what rate should you increase priorities? And, as system gets overloaded, no job gets CPU time, so everyone increases in priority. The result is that interactive jobs suffer -- both short **and** long jobs have high priority!

Instead, **lottery scheduling:** give every job some number of lottery tickets, and on each time slice, randomly pick a winning ticket. On average, CPU time is proportional to # of tickets given to each job.

How do you assign tickets? To approximate SRTCF, short running jobs get more, long running jobs get fewer. To avoid starvation, every job gets at least one ticket (so everyone makes progress).

Major advantage over strict priority scheduling: behaves gracefully as load changes. Adding or deleting a job affects all jobs proportionately, independent of how many tickets each job has. For example, if short jobs get 10 tickets, and long jobs get 1 each, then:

# short jobs / # long jobs	% of CPU each short job gets	% of CPU each long job gets
1 / 1	91%	9%

0 / 2	NA	50%
2 / 0	50%	NA
1 0 / 1	10%	1%
1 / 1 0	50%	5%