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)

2. **Maximize throughput**: operations (or jobs) per second

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

Tradeoff: will argue you can get better average response time by making system **less** fair.

11.2 Assumptions

We'll 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: what happens if you remove these constraints?

11.3 Scheduling policies:

11.3.1 FIFO

Different names for the same thing:

FCFS -- first come first serve
FIFO -- first in first out
Run until done

In early systems, FIFO => one program kept CPU until it completely finished.

For this course, FIFO => keep CPU until thread blocks

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

11.3.2 Round Robin

Round-robin: after time slice, move thread to back of the queue

11.3.2.1 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 ~ 10 milliseconds; typical context switch takes 0.1 millisecond, so roughly 1% overhead due to time-slicing.

11.3.2.2 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 of 1 second. All jobs start at the same time.

Job completion times

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

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

11.3.3 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.

An easier acronym to remember: **SJF**: shortest job first. Means same thing as STCF and SRTCF (non-preemptive SJF and preemptive SJF).

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.

11.3.3.1 Comparison of SJF with FIFO and Round Robin

What if all jobs are the same length?

SJF is the same as FIFO

What if jobs have varying length?

SJF (and round robin): short jobs don't get stuck behind long jobs.

Example to illustrate the benefits of preemptive SJF:

Three jobs:

A, B: both CPU bound, run for week

C: I/O bound, loop

1 ms of CPU

10 ms of disk I/O

What if we run C by itself? A or B by itself?

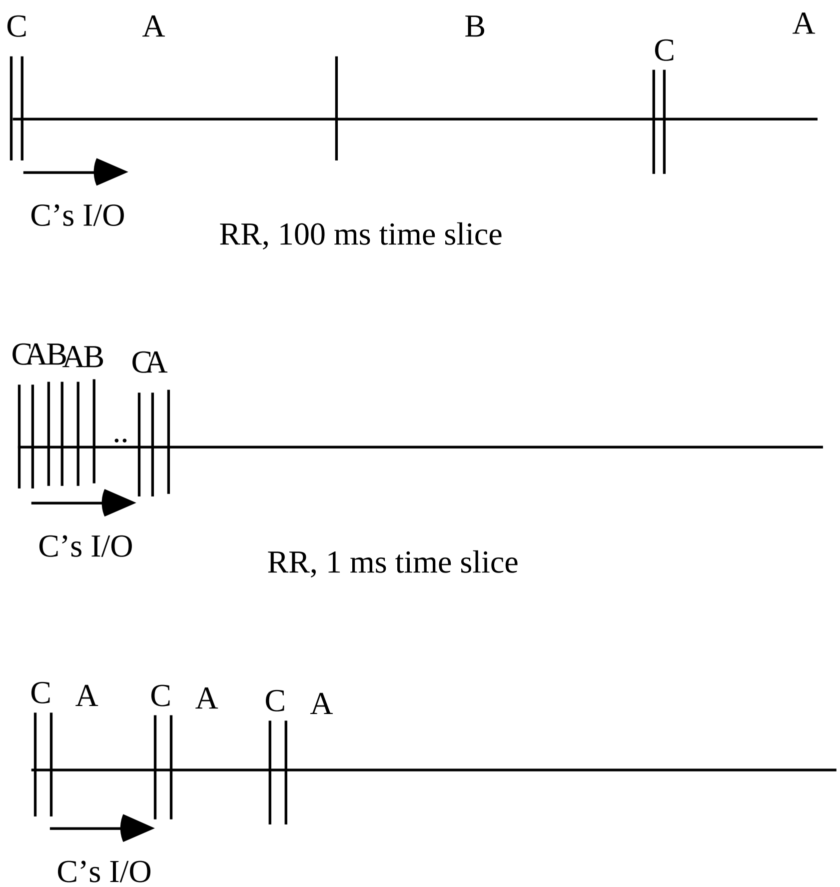
What if we run all three jobs together?

With FIFO? What about non-preemptive SJF (STCF)?

With Round Robin (100 ms time slice)?

With Round Robin (1 ms time slice)?

With preemptive SJF (SRTCF)?



SRCTF

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

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

SJF pros&cons:

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

11.3.3.2 Knowledge of future

Problem: SJF requires knowledge of the future.

How do you know how long program will run for?

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

To stop cheating, if your job takes more than what you said, system kills your job.

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

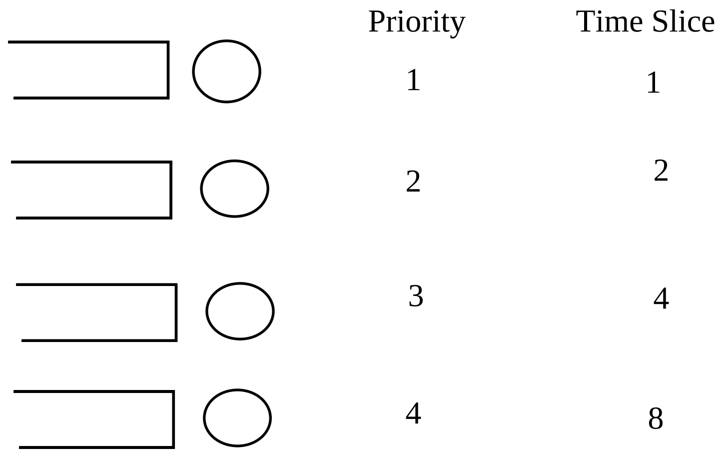
11.3.4 Multilevel feedback

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 SJF!

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

Multilevel feedback queues (like SJF) 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!

11.3.5 Lottery scheduling

What should we do about fairness? Since SJF 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.

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!

Recent idea: **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 SJF, short jobs get more tickets, long running jobs get fewer. To avoid starvation, every job gets at least one ticket (so everyone makes progress).

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
10/1	10%	1%
1/10	50%	5%

11.4 Queueing Theory

Question: when should you buy a faster computer?

One approach: Buy when it will pay for itself in improved response time (since you pay for worse response time in reduced productivity, customer angst, ...)

Queueing theory allows you to **predict** how response time will change as a function of hypothetical changes in # of users, speed of the CPU, speed of the disk, etc.

Might think that you should buy a faster X when X is utilized 100% of the time. But for most systems, response time goes to infinity, as utilization goes to 100%.

How does response time vary with # of users?

Worst case: all users submit jobs at same time. Thus, response time gets linearly worse as we add extra users/linearly better as we make computer go faster.

Best case: each user submits job just as last one completes. As increase # of users, no impact on response time! This continues until system is completely utilized, then response time gets linearly worse.

What if we assume users submit jobs randomly, and they take random amounts of time? Possible to show mathematically (fine print: assuming exponential distribution):

response time = service time / (1 - utilization)