Lecture #3: CPU Scheduling

****
Review -- 1 min
****

Deadlock
- ♦ definition
- ♦ conditions for its occurrence
- ♦ solutions: breaking deadlocks, avoiding deadlocks
- ♦ efficiency v. complexity

Other hard (liveness) problems
- ■ priority inversion
- ■ starvation
- ■ denial of service

****
Outline - 1 min
****

CPU Scheduling
- goals
- algorithms and evaluation

Goal of lecture:
We will discuss a range of options. There are many more out there.
The important thing is not to memorize the scheduling algorithms I describe.
The important thing is to develop strategy for analyzing scheduling algorithms in general.

****
Preview - 1 min
****

File systems

****

Lecture - 20 min

****

1. Scheduling problem definition

Threads = concurrency abstraction

Last several weeks: what threads are, how to build them, how to use them

3 main states: ready, running, waiting

- Running: TCB on CPU
- Waiting: TCB on a lock, semaphore, or condition variable queue
- Ready: TCB on ready queue
-

Operating system can choose when to stop running one and when to start the next ready one. OS can choose which ready thread to start (ready "queue" doesn't have to be FIFO)

Key principle of OS design: separate **mechanism** from **policy**

Mechanism – how to do something

Policy – what to do, when to do it

In this case, design our **context switch mechanism** and **synchronization methodology** allow OS to switch from any thread to any other one **at any time** (system will behave correctly)

Thread/process scheduling policy decides when to switch in order to meet performance goals

1.1 Pre-emptive v. non-preemptive

Non-preemptive – once a process starts, it is allowed to run until it finishes (or gives up CPU by calling "yield()" or wait())

- simple and efficient to implement
- creates problems (what are they? How to solve?)

Pre-emptive – process switched between "ready" and "running" state
→ timer can cause context switch

- more sophisticated and powerful
- less efficient (more context switches)

2. Scheduling policy goals

Step 1 in choosing a good policy is deciding on your goals:
Today case study: balance 3 goals

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

Response time = average (process end time – process start time)

NOTE: THIS DEFINITION DIFFERS FROM THE ONE IN THE BOOK!

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

CPU utilization = time CPU is doing useful work/total elapsed time

Two parts to max throughput

a) minimize overhead
context switch overhead – the time two switch between threads
(handle interrupt, copy state, flush/reload caches, …)
Note: b/c of context switch overhead, increasing frequency of context switches may reduce throughput

b) efficient use of system resources (not only CPU, but also disk, memory, etc)

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

What does fairness mean?

Fairness is interpreted in context of **priorities** -- if user says job A is more important than job B, is it fair to give job A more resources than B? (Yes.)

Minimal definition of fairness: freedom from starvation
Starvation – indefinite blocking
Starvation free -- system **guarantees** that if a job j is **ready**, eventually it will run (regardless of workload/what new jobs arrive)

Fairness v. minimize response time –fairness is sometimes a tradeoff against average response time. You can get sometimes get better average response time by making system **less** fair

Note:
1) First step in evaluating policy is to pick goals
2) Goals can be in conflict (challenge in picking policy is evaluating trade-offs among goals for a workload)
3) Today look at 3 goals, but other goals exist:

QUESTION: Other goals?

- real time

- predictable
- …

See vin's notes, book for more details for policies evaluation of different algorithms under these goals.

3. Scheduling doesn't matter (usually)

[at least not for response time, throughput. Ignore following for priority...]

load v. response time curve
(A little bit of queing theory)

r = 1/(1-u) (exponential arrivals)

-->
When load low, scheduling doesn't matter
When load high, scheduling doesn't matter
When load medium, scheduling matters (but if load, hardware, application change, scheduling stops mattering)
...
[also, for later discussion, when requests are small, equally important, and similar size, scheduling doesn't m atter]

--> lots of servers do FIFO and are done...

What does matter?

4. Overload control

Servers often have highly variable load
-- 4:1 typical
-- flash crowd (e.g., UT emergency response server)
-- ebay auction last few minutes
-- ticketmaster -- high-profile event goes on sale...

May be impossible (or frightfully expensive) to provision system to eliminate possiblity of overload

overload often happens when service matters most!

--> Must design servers to control overload

Can't solve problem with scheduling.

Solution 1: reduce work
-- Distasteful answer, but sometimes necessary
-- e.g., server with FIFO queue of work; make queue finite size

Examples:
-- reject requests (ugh, but simple)
-- do less work per request
-- e.g., switch from 720p to 480i video feed, serve static version of CNN front page rather than dynamically-generated (give up advertising revenue to survive crisis, slow down how frequenlty auction prices updated, ...)

e.g., ebay
http://www.cs.cornell.edu/courses/cs5410/2008fa/Slides/eBay.pdf

-- Turn off other services
-- e.g., mail server provides access to mailboxes, but queues outgoing and incoming mail

NOTE: "Reduce work" can require careful design
e.g., de facto way to limit load is to let network accept queue fill, but that is horrible for users
e.g., stage 5 drops work when server overloaded --> lots of work but nothing gets done

[MDD: TBD: look at receiver livelock paper?]

Solution 2: increase resources

"Cloud" *may* allow services to grow/shrink as demands change
--> cost proportional to resources used v. cost proportional to peak

[hope is that cloud is big enough with diverse enough demands that the subset of services with high demand can get the resources they need]

jury is out on this...

5. Scheduling policies

How to evaluate policies?

5.1 FIFO

different names for same thing
FCFS – first come first serve
FIFO – first in first out
Run until done

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

Later, FIFO means: keep CPU until thread blocks (goes to a "waiting" queue)

I'll assume this

QUESTION: Response time, throughput, fairness
FIFO pros&cons
+ simple
+ no starvation
+ few context switches
- short jobs get stuck behind long jobs

EXAMPLE

5.2 Round Robin

Solution? Add timer, and preempt CPU from long-running jobs.

Just about every real OS does something of this flavor.

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

Response time v throughput

5.2.1 How do you choose the time slice?

1) what if too big?
Response time suffers

2) what if too small?
Throughput suffers. Spend all of 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 context switch is .1-1ms, so roughly 1% of time is time-slice overhead

5.2.2 Comparison between FIFO and Round Robin

QUESTION: Assuming zero-cost context switch overhead, is RR always better than FIFO?

No. Counterexample: 10 jobs, each takes 100 seconds of CPU time.
Round robin time slice of 1 second.
All start at same time

Job Completion Time

Job #	FIFO	RR
1	100	991
2	200	992
3	300	993
…		
9	900	999
10	1000	1000

Round robin runs one second from each job, before going back to 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

QUESTION: Response time, throughput, fairness

Thus, RR pros&cons
+ Fairness: In some sense it is fair – each job gets equal shot at CPU
o Throughput: shorter time slices increase overhead
  → make time slice large compared to context switch overhead
Response time:
+ better for short jobs (and not too bad for long) when jobs are mixed length
- poor when jobs are same length (and longer than time slice)

5.3 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 to get short jobs out of the system
Big effect on short jobs, small effect on large jobs.
Result – better average response time

Example: copier machine

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 are optimal.

> Intuition: start with a STCF schedule. Swap any two jobs – A, B -- on the schedule. Any job before A_orig or after B_orig will complete at same time. A will now finish when B would have finished. But B will finish later than A would have finished (and all jobs between A and B will finish later than they would have finished.)

Since SRTCF is always at least as good as STCF, focus on SRTCF.

5.3.1 Comparison of SRTCF with FIFO and RR

What if all jobs are 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) are better than FIFO – short jobs don't get stuck behind long jobs

Example to illustrate SRTCF:

3 jobs
A, B: both CPU bound, run for a week
C: I/O bound, loop

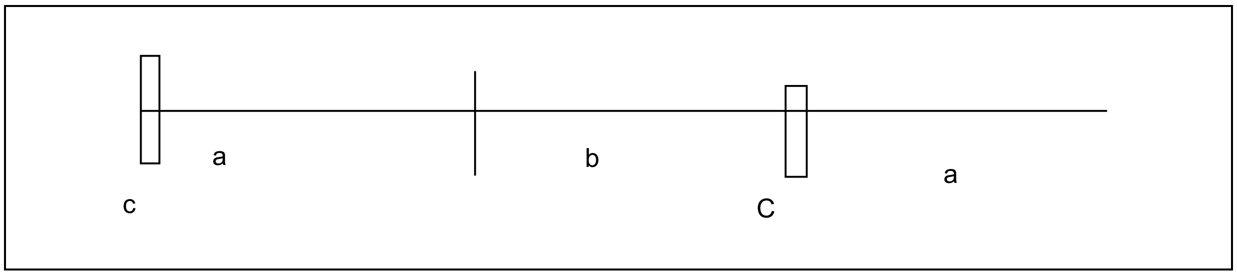
1ms of CPU
10ms of disk I/O

By itself, C uses 90% of disk
By itself, A or B uses 100% of CPU

What happens if try to share system between A, B, and C?

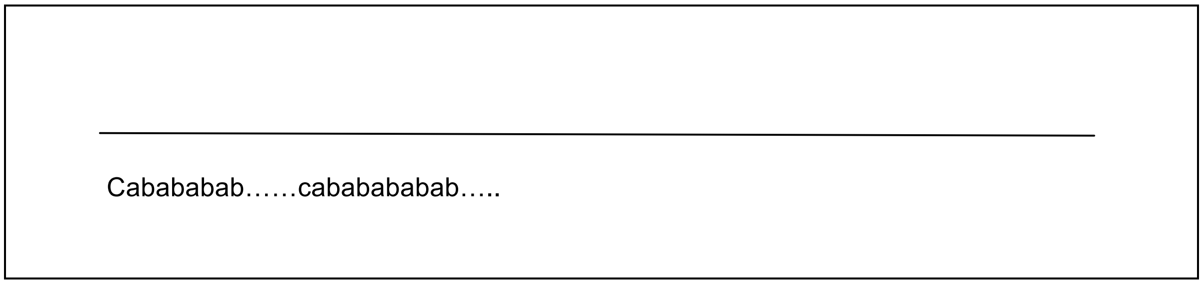
With FIFO:
once A or B gets in, keep CPU for 2 weeks

With Round Robin (100ms time slice)



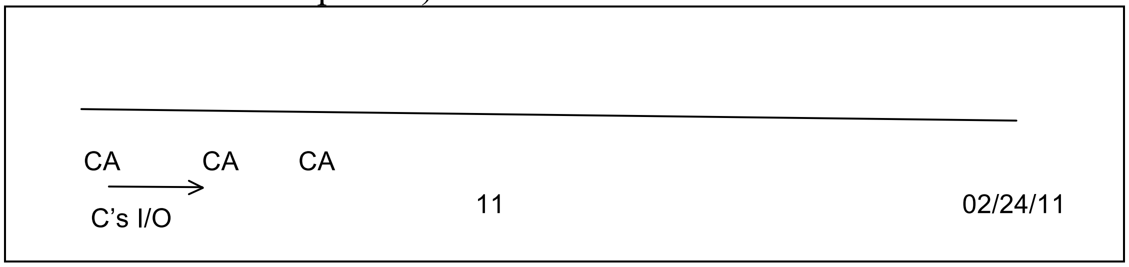
only get 5% disk utilization

With round robin (1ms time slice)



Get nearly 90% disk utilization; almost as good as C alone, but don’t slow A or B by that much; they still get 90% of CPU

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



QUESTION: when do A and B finish under RR (1ms) and SRTCF?

QUESTION: Response time, throughput, fairness

STRCF pros&cons
+ Response time: optimal (average response time)
+ Throughput: low overhead
- Fairness: can we get starvation?
A downside to SRTCF is that it can lead to starvation – lots of short jobs can keep long jobs from ever making progress

What is the biggest limitation?
- hard to predict the future!!

5.3.2 Knowledge of the future

Problem STCF/SRTCF require knowledge of the future

How do you know how long a program will run for?

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

(QUESTION: Running STRCF – what do you tell the system???)

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

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

(Good way to do CS development – figure out what the *right* answer is, then figure out how to approximate it)

5.4 Multilevel Feedback

Central idea in CS (occurs in lots of places) – **use past to predict future**. If program was I/O bound in the past, likely to be in the future

If computer behavior were random, history won't help
Or if past behavior is opposite of current behavior

Most of the time, though, program behavior is regular
How to exploit this?
If past behavior predicts future behavior, then favor jobs that have been using CPU the least amount of time to approximate SRTCF!

Adaptive policies – change policy based on past behavior.
Used in CPU scheduling, virtual memory, in file system …

Multilevel 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 second highest, etc
--round-robin time slices increase exponentially at lower priority

Queue	Prioiry	Time slice
XXXXXO	1	1
XXXXXO	2	2
XXXXXO	3	4
XXXXXO	4	8

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 1 level (or back to top)

QUESTION: Response time, throughput, fairness

Results approximate SRTCF: CPU bound jobs drop like a rock while short running I/O bound jobs stay near top

Multilevel feedback queues (like SRTCF) still unfair – long running jobs may never get the CPU
    QUESTION: How to solve?

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

5.5 Lottery scheduling

What should we do about fairness? Since SRTCF is optimal and unfair, any increase in fairness (e.g. giving long jobs a fraction of the CPU even when there are shorter jobs to run) will 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 1 long-running job and 100 short runing jobs?

Could adjust priority: increase priority of jobs as they **don't** get service. This is what UNIX does

Problem – this is ad hoc - at what rate should you increase priorities? And, as system gets overloaded, no job gets CPU time, so everyone increases in priority (→ shorter time slices; → less efficient just when system is busiest); also, interactive jobs suffer – both short and long jobs have high priority

Recent research (~1995-1997) – **proportional share schedulers** – allow scheduler to specify what fraction of resources go to each thread
Proportional share schedulers emphasize fairness as main goal

Several schedulers exist: start-time fair queuing (invented by Vin and students here at UT) is the best, stride scheduling is OK. But, I'll explain a simple one (that is not as good as SFQ or stride)

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 number of tickets to each job

How will lottery scheduling behave wrt latency?
Can we improve lottery scheduling to approximate multi-level feedback/SRTF?

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

Advantages over strict priority scheduling:
behaves gracefully as load changes – adding or deleting a job affects all jobs proportionally, independent of how many ticket each job has

For example, if short job gets 10 tickets and long gets 1 each then

#short/#long	%CPU per short	%Cpu per long
1/1	91%	9%
0/2	NA	50%
2/0	50%	N/A
10/1	10%	1%
1/10	50%	5%

6. A little queuing theory

Question: when should you buy a faster computer?
One approach – buy when it will pay for itself in improved response time

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

Might think you shouldn't buy a faster X when X has spare capacity (utilization of X < 100%), but for most systems, response time goes to infinity as utilization goes to 100%

How does response time vary with # users?

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

Best case: each user submits job after previous one completes.
As increase #users, no impact on response time (until system completely utilized)

What if we assume users submit jobs randomly and they take random amounts of time. Possible to show mathematically:

response time = service time / (1-utilization)

fine print – exponential distribution

Implications:

(1) If a load is bursty it will have worse avg response time than if it is even
(2) When load is low, scheduling may not matter
(3) When low is high, scheduling may not matter
- (Actually – when overloaded can't serve all requests, so instead of "scheduling" the problem is deciding which requests to service

and which requests to drop; this choice often does matter)

(4) When scheduling matters, increasing capacity a bit may be a better way to solve problem than clever scheduling

****

Summary - 1 min

****

3 meta-lessons in system design

1) Separate mechanism from policy
   In this case: thread *mechanism* should allow context switch at any time → we can use any policy we want

2) Know your goals
   Often, when you are talking about policy you are doing so b/c there is some sort of trade-off of one goal against another. Explicitly write down what your goals are, which is most important, …

   Today talked about response time (and throughput and fairness). Different algorithms when worrying about real time.

3) Compare against optimal (even if you don't know how to build optimal for real system)
   - Provides reference to compare against (don't waste your time if you are already at 99% of optimal)
   - Provides insight used to understand other algorithms ("under what circumstances will I not be optimal?")

   In this case: SRTF is optimal
   - ➔ we can design algorithms that approximate it
   - ➔ we know: impossible to be both optimal and fair