CSE 410
Computer Systems

Hal Perkins
Spring 2010
Lecture 17 – Process Scheduling
Readings and References

• Reading
  – Operating System Concepts
    • Chapter 5, Secs. 5.1-5.5
      – Skim math for cultural enrichment; we won’t have
time to go into scheduling theory
Scheduling

• In discussing processes and threads, we talked about context switching
  – an interrupt occurs (device completion, timer, …)
  – a thread causes an exception (a trap or a fault)
• We glossed over the choice of which thread is chosen to be run next
  – “some thread from the ready queue”
• This decision is called scheduling
  • context switching is a mechanism inside the OS
  • scheduling is a policy
Scheduling Goals

• Keep the CPU(s) busy
• Maximize throughput (“requests” per second)
• Minimize latency
  – Time between responses
  – Time for entire “job”
• Favor some particular class (foreground window, interactive vs CPU-bound)
• Avoid jitter (video)
• Keep the airplane in the sky 😊
• Be fair (no starvation or inversion)
• THESE MAY CONFLICT
Classes of Schedulers

- **Batch**
  - Throughput / utilization oriented
  - Example: audit inter-bank funds transfers each night, Pixar rendering
- **Interactive**
  - Response time oriented
- **Hard Real Time**
  - Deadline driven
  - Example: embedded systems (cars, airplanes, etc.)
- **Soft Real Time**
  - Video, TIVO, etc.
- **Parallel**
  - Speedup driven
  - Example: “space-shared” use of a 1000-processor machine for large simulations
- **Others…**

- We’ll be talking primarily about interactive schedulers (as does the text).
Multiple levels of scheduling decisions

• Long term
  – Should a new “job” be “initiated,” or should it be held?
  – typical of batch systems
  – what might cause you to make a “hold” decision?
• Medium term
  – Should a running program be temporarily marked as non-runnable (e.g., swapped out)?
• Short term
  – Which thread should get the CPU next? For how long?
  – Which I/O operation should be sent to the disk next?
  – On a multiprocessor:
    • should we attempt to coordinate the running of threads from the same address space in some way?
    • should we worry about cache state (processor affinity)?
Scheduling Goals I: Performance

• Many possible metrics / performance goals (which sometimes conflict)
  – maximize CPU utilization
  – maximize throughput (requests completed/sec)
  – minimize average response time (average time from submission of request to completion of response)
  – minimize average waiting time (average time from submission of request to start of execution)
  – minimize energy (joules per instruction) subject to some constraint (e.g., frames/second)
Scheduling Goals II: Fairness

• No single, compelling definition of “fair”
  – How to measure fairness?
    • Equal CPU consumption? (over what time scale?)
  – Fair per-user? per-process? per-thread?
  – What if one thread is CPU bound and one is IO bound?

• Sometimes the goal is to be unfair:
  – Explicitly favor some particular class of requests (priority system), but…
  – avoid starvation (be sure everyone gets at least some service)
The basic situation

Scheduling:
- Who to assign each resource to
- When to re-evaluate your decisions

Schedulable units  Resources
When to assign?

• Pre-emptive vs. non-preemptive schedulers
  – Non-preemptive
    • once you give somebody the green light, they’ve got it until they relinquish it
      – an I/O operation
      – allocation of memory in a system without swapping
  – Preemptive
    • you can re-visit a decision
      – setting the timer allows you to preempt the CPU from a thread even if it doesn’t relinquish it voluntarily
      – in any modern system, if you mark a program as non-runnable, its memory resources will eventually be re-allocated to others
  • Re-assignment always involves some overhead
    – Overhead doesn’t contribute to the goal of any scheduler

• We’ll assume “work conserving” policies
  – Never leave a resource idle when someone wants it
    • Why even mention this? When might it be useful to do something else?
Algorithm #1: FCFS/FIFO

- First-come first-served / First-in first-out (FCFS/FIFO)
  - schedule in the order that they arrive
  - “real-world” scheduling of people in (single) lines
    - supermarkets, bank tellers, McD’s, Starbucks …
      - (sometimes we separate job classes – DMV)
    - typically non-preemptive
      - no context switching at supermarket!
  - jobs treated equally, no starvation
    - In what sense is this “fair”?

- Sounds perfect!
  - in the real world, when does FCFS/FIFO work well?
    - even then, what’s it’s limitation?
  - and when does it work badly?
FCFS/FIFO example

<table>
<thead>
<tr>
<th>time</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>C</td>
<td>A</td>
</tr>
</tbody>
</table>

• Suppose the duration of A is 5, and the durations of B and C are each 1
  – average response time for schedule 1 (assuming A, B, and C all arrive at about time 0) is \((5+6+7)/3 = 18/3 = 6\)
  – average response time for schedule 2 is \((1+2+7)/3 = 10/3 = 3.3\)
  – consider also “elongation factor” – a “perceptual” measure:
    • Schedule 1: A is 5/5, B is 6/1, C is 7/1 (worst is 7, ave is 4.7)
    • Schedule 2: A is 7/5, B is 1/1, C is 2/1 (worst is 2, ave is 1.5)
FCFS/FIFO drawbacks

• Average response time can be lousy
  – small requests wait behind big ones
• May lead to poor utilization of other resources
  – if you send me on my way, I can go keep another resource busy
  – FCFS may result in poor overlap of CPU and I/O activity
Algorithm #2: SPT/SJF

- Shortest processing time first / Shortest job first (SPT/SJF)
  - choose the request with the smallest service requirement
- *Provably optimal* with respect to average response time
SPT/SJF optimality

- In any schedule that is not SPT/SJF, there is some adjacent pair of requests f and g where the service time (duration) of f, $s_f$, exceeds that of g, $s_g$
- The total contribution to average response time of f and g is $2t_k + 2s_f + s_g$
- If you interchange f and g, their total contribution will be $2t_k + 2s_g + s_f$, which is smaller because $s_g < s_f$
SPT/SJF drawbacks

• It’s non-preemptive
  – So?

• … but there’s a preemptive version – SRPT (Shortest Remaining Processing Time first) – that accommodates arrivals (rather than assuming all requests are initially available)

• Sounds perfect!
  – what about starvation?
  – can you know the processing time of a request?
  – can you guess/approximate? How?
Algorithm #3: RR

• Round Robin scheduling (RR)
  – ready queue is treated as a circular FIFO queue
  – each request is given a time slice, called a quantum
    • request executes for duration of quantum, or until it blocks
      – what signifies the end of a quantum?
    • time-division multiplexing (time-slicing)
  – great for timesharing
    • no starvation

• Sounds perfect!
  – how is RR an improvement over FCFS?
  – how is RR an improvement over SPT?
  – how is RR an approximation to SPT?
  – what are the warts?
RR drawbacks

- What if all jobs are exactly the same length?
  - What would the pessimal schedule be?

- What do you set the quantum to be?
  - no value is “correct”
    - if small, then context switch often, incurring high overhead
    - if large, then response time degrades
  - treats all jobs equally
    - how might we fix this?
Algorithm #4: Priority

- Assign priorities to requests
  - choose request with highest priority to run next
    - if tie, use another scheduling algorithm to break (e.g., RR)
    - to implement SJF, priority = expected length of CPU burst

- Abstractly modeled (and usually implemented) as multiple “priority queues”
  - put a ready request on the queue associated with its priority

- Sounds perfect!
Priority drawbacks

• How are you going to assign priorities?

• Starvation
  – if there is an endless supply of high priority jobs, no low-priority job will ever run

• Solution: “age” threads over time
  – increase priority as a function of accumulated wait time
  – decrease priority as a function of accumulated processing time
  – many ugly heuristics have been explored in this space
Combining algorithms

• In practice, any real system uses some sort of hybrid approach, with elements of FCFS, SPT, RR, and Priority

• Example: multi-level feedback queues (MLFQ)
  – there is a hierarchy of queues
  – there is a priority ordering among the queues
  – new requests enter the highest priority queue
  – each queue is scheduled RR
  – queues have different quanta
  – requests move between queues based on execution history
UNIX scheduling

- Canonical scheduler is pretty much MLFQ
  - 3-4 classes spanning ~170 priority levels
    - timesharing: lowest 60 priorities
    - system: middle 40 priorities
    - real-time: highest 60 priorities
  - priority scheduling across queues, RR within
    - thread with highest priority always run first
    - threads with same priority scheduled RR
  - threads dynamically change priority
    - increases over time if thread blocks before end of quantum
    - decreases if thread uses entire quantum
- Goals:
  - reward interactive behavior over CPU hogs
    - interactive jobs typically have short bursts of CPU
Summary

- Scheduling takes place at many levels
- It can make a huge difference in performance
  - this difference increases with the variability in service requirements
- Multiple goals, sometimes (always?) conflicting
- There are many “pure” algorithms, most with some drawbacks in practice – FCFS, SPT, RR, Priority
- Real systems use hybrids
- Scheduling is still important, particularly in large-scale data centers – for reasons of both cost and energy