Scheduling

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

Multiple levels of scheduling decisions

• Should a new “job” be “initiated,” or should it be held?
  – typical of batch systems, including modern scientific computing systems
  – what might cause you to make a “hold” decision?
• Should a program that has been running be temporarily marked as non-runnable (e.g., swapped out)?
• Which thread should be given 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?

Preemptive vs. non-preemptive scheduling

• 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
  • doesn’t really require swapping – in a virtual memory system, the page frames will get preempted, even though this isn’t the efficient way to do it

Scheduling goals

• Scheduling algorithms can have many different goals (which sometimes conflict)
  – maximize CPU utilization
  – maximize throughput (requests completed / s)
  – 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)
  – favor some particular class of requests (priority system)
  – avoid starvation (be sure everyone gets at least some service)

• Goals may depend on type of system
  – transaction processing system: strive to maximize throughput
  – interactive system: strive to minimize response time of interactive requests (e.g., editing, vs. compiling)
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 lines
    - supermarkets, bank tellers, McD’s, Starbucks …
  - typically non-preemptive
    - no context switching at supermarket!
  - jobs treated equally, no starvation

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

FCFS example

- 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)

SPT/SJF optimality

- In any schedule that is not SPT, 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 $2s_f + 2s_g$
- If you interchange f and g, their total contribution will be $2s_g + 2s_f$, which is smaller because $s_g < s_f$
- If the variability of request durations is zero, how does FCFS compare to SPT for average response time?

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 drawbacks

- It’s non-preemptive … 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?
  - what are the warts?

RR drawbacks

- 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
  - if I run 100 copies of SETI@home, it degrades your service
  - how might I 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., FCFS)
  - 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
    - process with highest priority always run first
    - processes with same priority scheduled RR
  - processes dynamically change priority
  - increases over time if process blocks before end of quantum
  - decreases if process uses entire quantum
- Goals:
  - reward interactive behavior over CPU hogs
  - interactive jobs typically have short bursts of CPU
Scheduling the Apache web server SRPT

- What does a web request consist of? (What’s it trying to get done?)
- How are incoming web requests scheduled, in practice?
- How might you estimate the service time of an incoming request
- Starvation is a problem in theory – is it a problem in practice?
  - “Kleinrock’s conservation law”

(Recent work by Bianca Schroeder and Mor Harchol-Balter at CMU)

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 conflicting
- There are many “pure” algorithms, most with some drawbacks in practice – FCFS, SPT, RR, Priority
- Real systems use hybrids
- Recent work has shown that SPT/SRPT – always known to be beneficial in principle – may be more practical in some settings than long thought