CSE 451: Operating Systems
Winter 2005

Lecture 6
Scheduling

Steve Gribble

Scheduling

- In discussion process management, we talked about context switching between threads/process on the ready queue
  - but, we glossed over the details of which process or thread is chosen next
  - making this decision is called scheduling
    - scheduling is policy
    - context switching is mechanism
- Today, we'll look at:
  - the goals of scheduling
    - starvation
  - well-known scheduling algorithms
    - standard UNIX scheduling

Multiprogramming and Scheduling

- Multiprogramming increases resource utilization and job throughput by overlapping I/O and CPU
  - today: look at scheduling policies
    - which process/thread to run, and for how long
  - schedulable entities are usually called jobs
    - processes, threads, people, disk arm movements, …
- There are two time scales of scheduling the CPU:
  - long term: determining the multiprogramming level
    - how many jobs are loaded into primary memory
    - act of loading in a new job (or loading one out) is swapping
  - short-term: which job to run next to result in "good service"
    - happens frequently, want to minimize context-switch overhead
    - good service could mean many things

Scheduling

- The scheduler is the module that moves jobs from queue to queue
  - the scheduling algorithm determines which job(s) are chosen to run next, and which queues they should wait on
  - the scheduler is typically run when:
    - a job switches from running to waiting
    - when an interrupt occurs
      - especially a timer interrupt
    - when a job is created or terminated
- There are two major classes of scheduling systems
  - in preemptive systems, the scheduler can interrupt a job and force a context switch
  - in non-preemptive systems, the scheduler waits for the running job to explicitly (voluntarily) block

Scheduling Goals

- Scheduling algorithms can have many different goals (which sometimes conflict)
  - maximize CPU utilization
  - maximize job throughput (#jobs/s)
  - minimize job turnaround time ($T_{finish} - T_{start}$)
  - minimize job waiting time ($Avg(T_{wait})$: average time spent on wait queue)
  - minimize response time ($Avg(T_{resp})$: average time spent on ready queue)
- Goals may depend on type of system
  - batch system: strive to maximize job throughput and minimize turnaround time
  - interactive systems: minimize response time of interactive jobs (such as editors or web browsers)

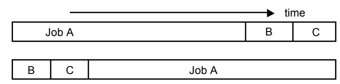
Scheduler Non-goals

- Schedulers typically try to prevent starvation
  - starvation occurs when a process is prevented from making progress, because another process has a resource it needs
- A poor scheduling policy can cause starvation
  - e.g., if a high-priority process always prevents a low-priority process from running on the CPU
- Synchronization can also cause starvation
  - we'll see this next class
  - roughly, if somebody else always gets a lock I need, I can't make progress

Algorithm #1: FCFS/FIFO

- First-come first-served (FCFS)
  - jobs are scheduled in the order that they arrive
  - “real-world” scheduling of people in lines
    - e.g. supermarket, bank tellers, MacDonalds, …
  - typically non-preemptive
    - no context switching at supermarket!
  - jobs treated equally, no starvation
    - except possibly for infinitely long jobs
- Sounds perfect!
  - what’s the problem?

FCFS picture



- Problems:
  - average response time and turnaround time can be large
    - e.g., small jobs waiting behind long ones
    - results in high turnaround time
  - may lead to poor overlap of I/O and CPU

Algorithm #2: SJF

- Shortest job first (SJF)
  - choose the job with the smallest expected CPU burst
  - can prove that this has optimal min. average waiting time
- Can be preemptive or non-preemptive
  - preemptive is called shortest remaining time first (SRTF)
- Sounds perfect!
  - what’s the problem here?

SJF Problem

- Problem: impossible to know size of future CPU burst
  - from your theory class, equivalent to the halting problem
  - can you make a reasonable guess?
    - yes, for instance looking at past as predictor of future
    - but, might lead to starvation in some cases!

Priority Scheduling

- Assign priorities to jobs
  - choose job 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 as multiple “priority queues”
  - put ready job on queue associated with its priority
- Sound perfect!
  - what’s wrong with this?

Priority Scheduling: problem

- The problem: starvation
  - if there is an endless supply of high priority jobs, no low-priority job will ever run
- Solution: “age” processes over time
  - increase priority as a function of wait time
  - decrease priority as a function of CPU time
  - many ugly heuristics have been explored in this space

Round Robin

- Round Robin scheduling (RR)
  - ready queue is treated as a circular FIFO queue
  - each job is given a time slice, called a quantum
    - job executes for duration of quantum, or until it blocks
    - time-division multiplexing (time-slicing)
  - great for timesharing
    - no starvation
    - can be preemptive or non-preemptive
- Sounds perfect!
  - what's wrong with this?

RR problems

- Problems:
  - what do you set the quantum to be?
    - no setting is "correct"
      - if small, then context switch often, incurring high overhead
      - if large, then response time drops
  - treats all jobs equally
    - if I run 100 copies of SETI@home, it degrades your service
    - how can I fix this?

Combining algorithms

- Scheduling algorithms can be combined in practice
  - have multiple queues
  - pick a different algorithm for each queue
  - and maybe, move processes between queues
- Example: multi-level feedback queues (MLFQ)
  - multiple queues representing different job types
    - batch, interactive, system, CPU-bound, etc.
  - queues have priorities
    - schedule jobs within a queue using RR
  - jobs move between queues based on execution history
    - "feedback": switch from CPU-bound to interactive behavior
- Pop-quiz:
  - is MLFQ starvation-free?

UNIX Scheduling

- Canonical scheduler uses a MLFQ
  - 3-4 classes spanning ~170 priority levels
    - timesharing: first 60 priorities
    - system: next 40 priorities
    - real-time: next 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