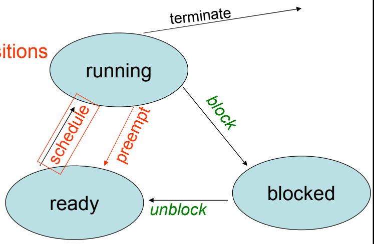
Recall from last week…

- Process states
  - scheduler transitions
    - (red)



- Challenges:
  - Which process should run?
  - When should processes be preempted?
  - When are scheduling decisions made?

Today: Process Scheduling Algorithms

- Objective: a high performance system
  - Efficiency:
    - Maximize CPU time spent executing user programs.
    - Recall that context switch is expensive.
      - on the order of 10^4 instructions
    - But not at the expense of….
  - Responsiveness
    - What do I mean by responsiveness?
      - Average user happiest?
      - Long computations complete in reasonable time?
- Several approaches will be described.

Process Scheduling by Objective

Eric Freudenthal

(almost) Universal scheduler algorithm

- Run process with highest “priority”
  - Computed priority represents some scheduling objective
  - Priority can only be computed from available information
    - Assigned process importance (if available)
    - How long in ready queue, how long running
    - Characteristics of process
      - i/o bound, resources held, how long since submission
- When does scheduler algorithm execute?
  - Whenever running process blocks
  - Maybe at other times too:
    - Maybe: Whenever a process becomes ready.
    - Maybe: Whenever quantum expires.
      - Is quantum fixed? If not, how is it computed?
- Challenge: mapping objectives to priorities

Name Game Warning Play at your own risk.

- The algorithms described today are known by multiple names.
- I use names that appear in Tannenbaum.
  - Allan assures me that his exams will use the names (not acronyms) **as they appear in Tannnenbaum**.
- Allan’s class notes include a table titled “the name game” listing the algorithms’ names in multiple text books.

Objective: Fairness (first attempt): **First-Come First-Served**

- Process that has been “ready” the longest has highest priority.
  - Head item if “ready queue” is a FIFO
- No preemption
  - Processes execute until they terminate or block.
- A process can “hog” the processor, starving others.

Objective: Fairness
Round Robin

First-Come First-Served with Preemption

- Preempt processes that 'hog' the processor
  - How to pick quantum
    - Extreme fairness: q = 1 instruction
      - Cost of context switching consumes >99.9% of CPU
    - Reasonable q = 1ms = 0.001s
      - Modern processors execute Approx 1G i/s
      - 1M instructions = (approx) 1ms
      - Approx 1/1,000,000 of cpu time lost due to preemption

Variants on **Round Robin**

- Prioritization by adjusting the quantum
  - Is it "fairer" to provide more execution time to some processes:
    - Those holding resources that effectively delay others
    - Those pay more?
    - Maybe: increase q for these "higher priority" processes.
- All processes have quantum = ∞
  - No preemption, therefore "First come first served"

Theoretical digression:
Processor Sharing

- This is a theoretical model
  - Each of n ready processes proceeds at rate 1/n.
  - For example, if 3 processes are ready, each executes 1/3 of an instruction in 1 cycle.
  - Useful for mathematical analysis since it models a process' *effective rate of execution* as a fraction.
- As if RR could have tiny quantum
  - (say 0.0001i)

Objective: ***important*** processes proceed most quickly

Priority Scheduling

- Processes assigned rank at entry.
  - Perhaps users pay more for higher rank?
- Process with highest "rank" always runs.
  - Round-robin if multiple at highest rank
- Preemption:
  - Run scheduler every time a process becomes ready.
    - preempt if higher rank process is ready
- Two challenges: starvation and priority inversion. (next two slides)

Priority challenge 1:
Starvation

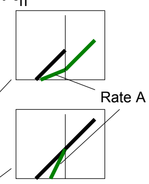
- Problem:
  - Low priority process may never run
- Solution: Priority aging
  - Temporarily raise rank of ready processes at some rate.
  - Effect: processes with lower rank wait longer to run if higher priority processes are ready.
  - When is aging computation performed?
    - When processes become ready.
    - When quantum expires

Priority Challenge 2:
"Priority inversion" possible

- Low rank process holds resource needed by high rank process.
  - Example
    - A: rank = 3, needs tape drive (blocked)
    - B. rank = 2, ready
    - C: rank = 1, has tape drive, ready
- Problem:
  - B has higher rank than C
  - So B will execute, and A will be delayed.
    - Effectively inverts priority!!!!
- Solution: temporary "promote" C to A's priority:
  - Promotion rule: All low rank processes {C} holding resource req'd by some higher rank process A, are temporarily promoted to A's rank.

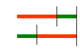
Objective: giving older jobs advantage: Selfish **Round-Robin**

- Round-robin among the 'in' group of *accepted* processes.
- Really just a computed-rank algorithm.
- Every process π has increasing rank R_π
  - R_π initially zero
    - *Define acceptance threshold* T = max(R_π)
    - If R_π = T, π's state is *accepted*
      - Accepted processes scheduled using RR
  - R_π increases after arrival:
    - If R_π < T, increase V_π at rate "A"
    - If R_π = T, increase V_π at rate "B"
  - If B ≥ A, then monoprogrammed
  - If B = 0, then RR (since T = 0)
  - If A > B > 0, then new processes excluded for a while



Objective: Minimize waiting **Shortest Job First**

- Rank = -(remaining execution time)
- Minimizes waiting time
  - Consider two jobs A > B that never block
    - If A run before B, total waiting time = A + (A+B)
    - **If B run before A, total waiting time = B + (A+B)**
    - **True for more than two processes too.**
- Challenge: prior knowledge of execution time.
  - Reasonable variant: prioritize by burst length, and use past behavior to predict the future.
- Challenge: Starvation of long jobs.
  - "Solution": Priority aging
- Also: Preemptive version
  - **P**SJF – **preemptive** shortest job first
  - Shortest job remains shortest **if no shorter job becomes ready**



Fairness revisited: Prioritize disadvantaged processes.

- **Highest Penalty Ratio Next**
- Define "Penalty Ratio"
  - T = wall clock time since arrival
  - t = execution time
  - Penalty ratio r = T/t, highest r has priority
    - Represents how much process's progress has been penalized due to i/o and multiprogramming.
  - Nuisance: ratio undefined until run (fudge this)
- Preemptive variant:
  - Re-evaluate penalty ratios when processes unblock
  - Set timer to expire when current process no longer highest priority
  - Be careful not to allow timer period to approach zero!

Objective: Favor Interactive Processes

Multi-Level Queues

- Multiple classes of processes
  - Class 3: Interactive
  - Class 2: Batch
  - Class 1: Cycle-soaker (low priority background).
- Can be implemented using 3 queues
  - Policy among queues
    - For example: Run process with highest priority in highest non-empty queue.
  - Differing queues can implement different policies
    - For example, queue 1 could be FCFS

Favoring Interactive Processes with automatic detection.

Multi-level Feedback Queues

- An interactive process that doesn't block for a long time is demoted to 'background' and therefore treated differently (given lower priority…).
- A background process that blocks frequently can be promoted to interactive.
- Implemented using multilevel queues.
  - processes migrate between queues based on their recent behavior.

Questions?

- First Come First Served (no quantum)
- Round Robin (quantum)
  - Selfish Round Robin (snobish RR, latecomers wait)
  - Processor Sharing (theoretical RR)
- Priority Scheduling (highest priority runs)
  - Remember priority inversion!
- (preemptive) Shortest Job First
- Highest "Penalty Ratio" Next (greatest T/t)
- Multi-level Queues (distinct classes of job)
  - Multi-level Feedback Queues (auto classify)