Benjamin Fenster
CS-201 (Arslan)
6 November 2005

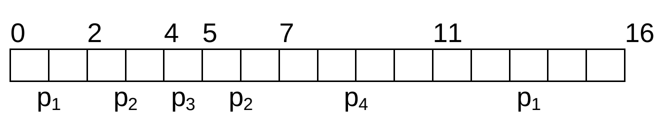
Scheduling

I. Introduction
  a. Concept: When the CPU is idle we want to give it more work. Of course, there are various objectives to consider
  b. Example Solution
    i. load, store, add instructions → CPU Burst (use CPU alone)
    ii. read from file → I/O burst
    iii. Then does more CPU instructions → CPU burst
    iv. Et cetera.
    v. This is typical for any process. Some processes are CPU bound, others I/O bound. Typical processes are mixed.
  c. Scheduler
    i. Selects a process from the ready queue and the CPU is allocated to that process.
    ii. The decision takes place when a process switches from running state to waiting state (i.e. makes an I/O request) or from waiting to ready (or terminates)
  d. Dispatcher
    i. Gives control of the CPU to the process selected by the scheduler.
    ii. This involves switching contest to the new process
      1. Switch to user mode
      2. Set PC to next instruction in the new process
    iii. Dispatch Latency: Time to stop one process and start another
  e. Scheduling Criteria
    i. CPU Utilization: Percent of time CPU is busy (maximize)
    ii. Throughput: Number of finished processes / total time (maximize)
    iii. Turnaround time: Amount of time to execute a process to completion (minimize)
    iv. Waiting Time: Time process has been waitin gin the ready queue (minimize)

II. First Come First Served (FCFS)
  a. P_1 (24), P_2 (3), P_3 (3) – Burst times; assume all start at time 0
  b. P_1 arrives first so it's executed first.
  c. Waiting Times: $P_1 = 0$, $P_2 = 24$, $p_3 = 27$.
  d. Average waiting time = 51 / 3 = 17
    i. Would be different if processes arrived in a different order
    ii. P_2, P_3, P_1, average = 0/3 = 3
    iii. AWT will be terrible if long jobs arrive first even if they're followed by short jobs

III. Shortest Job First (SJF)
  a. P_1 (7), P_2 (4), P_3 (1)
  b. Arriving at 0, 2, 4, 5
  c. Average Waiting Time: $p_0 = 0$. $p_2 = 8 – 2 = 6$. $p_3 = 7 – 4 = 3$. $p_4 = 12 – 5 = 7$.
  d. Average Waiting Time = (0 + 6 + 3 + 7) / 4 = 4
  e. Can *prove* that SJF gives the shortest waiting time.
  f. May leave a long process "starving," however, if short processes keep arriving.
  g. One possible improvement: stop a process and do something else for a while

IV. Preemptive Shortest Job First ("Shortest Remaining Time First")
  a. The algorithms so far have all been non-preemptive. Processes aren't interrupted to run other processes.
  b.



  c. AWT = (0 + 1 + 2 + 9) / 4 = 3

V. Determining CPU Burst Time
  a. These algorithms assume we know the next burst time!

b. We really have no idea in advance. Even the same loop may execute a different number of iterations so we cannot guess the burst time.
c. We'll use the past to predict the future.
d. Determine the length of the next CPU burst as: $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n$
    i. t_n= actual length of the nth CPU burst
    ii. τ_n = predicted length of the nth CPU burst.
    iii. When $\alpha = 0$, means $\tau_{n+1} = \tau_n$ so we just reuse the old prediction.
    iv. When $\alpha = 1$, $\tau_{n+1} = t_n$. Assume the next burst will take the same time as the last.
    v. $\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha\, t_{n-1} + \ldots + (1 - \alpha)^j \alpha\, t_{n-j} + (1 - \alpha)^{n+1}\, \tau_0$

VI. Priority Scheduling
a. Associate a priority number with every process.
b. The next process to be executed is the one with the highest priority.
c. Shortest job first could be implemented in this algorithm by just giving shorter jobs higher priority.
d. If jobs have the same priority, either develop a technique to resolve that or just let one be chosen at random.
e. Problem: Starvation. Low priority jobs may never execute if high priority jobs keep coming up.
    i. Could periodically increase priorities for jobs that are waiting. Called "aging"
    ii. Could also decrease processes' priorities as they execute.
f. Can be preemptive. If a new, high-priority process appears, stop the current process.

VII. Round-Robin (RR)
a. Have a simple queue
b. Each process executes for a certain time quantum.
c. enqueue it, then dequeue the next and execute that one.
d. The OS may switch to another process early if the current process requests IO, syscalls a wait, et cetera.
e. If the quantum is q and there are n processes, every process will wait q(n – 1) to execute each round (or rather: it won't wait *longer* than that).
    i. If q is small, lose more time to overhead for context switching. Performance may degrade if q is too small.
    ii. If q is large, processes wait longer. Response time and waiting time increase. If q is too large (infinite) you get the FCFS algorithm!
f. This is fair to all processes.
g. Throughput is likely worse than SJF (since this RR algorithm causes more context switches)

VIII. Multi-Level Queue
a. Have several queues.
b. Determine which queue a new process should join based on priority.
c. In each queue scheduling may be done separately.
d. Aging may be done by periodically moving processes into a higher priority queue.
e. Can add another scheduling component that determines when to execute processes from each queue (e.g. 80% from highest priority, 20% distributed among others)