CPU Scheduling

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
CPU Scheduling

• A policy deciding which process to run next, given a set of runnable processes
  – Happens frequently, hence should be fast

• Mechanism
  – How to transition?

• Policy
  – When to transition?
Basic Approaches

• Non-preemptive scheduling
  – The scheduler waits for the running process to voluntarily yield the CPU
  – Processes should be cooperative

• Preemptive scheduling
  – The scheduler can interrupt a process and force a context switch
  – What happens
    • If a process is preempted in the midst of updating the shared data?
    • If a process in a system call is preempted?
Terminologies

• Workload
  – A set of job descriptions
  – e.g. arrival time, run time, etc.

• Scheduler
  – A logic that decides when jobs run

• Metric
  – Measurement of scheduling quality
  – e.g. turnaround time, response time, fairness, etc.
Workload Assumptions

1. Each job runs for the same amount of time
2. All jobs arrive at the same time
3. Once started, each job runs to completion
4. All jobs only use the CPU (no I/O)
5. The run time of each job is known

• Metric: Turnaround time

\[ T_{\text{turnaround}} = T_{\text{completion}} - T_{\text{arrival}} \]
• First-Come, First-Served
  – Jobs are scheduled in order that they arrive
  – “Real-world” scheduling of people in lines
    • e.g. supermarket, bank tellers, McDonalds, etc.
  – Non-preemptive
  – Jobs are treated equally: no starvation

• Problems
  – Convoy effect
    • Average turnaround time can be large if small jobs wait behind long ones
SJF

• Shortest Job First
  – Each job has a variable run time (Assumption 1 relaxed)
  – Choose the job with the smallest run time
  – Can prove that SJF shows the optimal turnaround time under our assumptions
  – Non-preemptive

• Problems
  – Not optimal when jobs arrive at any time
  – Can potentially starve
FIFO vs. SJF

• FIFO

\[ T_{\text{turnaround}} = \frac{10 + 20 + 30}{3} = 20 \]

\[ T_{\text{turnaround}} = \frac{100 + 110 + 120}{3} = 110 \]

• SJF

\[ T_{\text{turnaround}} = \frac{10 + 20 + 120}{3} = 50 \]

\[ T_{\text{turnaround}} = \frac{100 + 90 + 100}{3} = 96.7 \]
STCF

• **Shortest Time-to-Completion First**
  – Jobs are not available simultaneously (Assumption 2 relaxed)
  – Preemptive version of SJF (Assumption 3 relaxed)
  – If a new job arrives with the run time less than the remaining time of the current job, preempt it

\[
A(100) \quad B(10), \quad C(10)
\]

\[
\text{T}_{\text{turnaround}} = \frac{(100 + 90 + 100)}{3} = 96.7
\]

**SJF**

\[
A(100) \quad B(10), \quad C(10)
\]

\[
\text{T}_{\text{turnaround}} = \frac{(120 + 10 + 20)}{3} = 50
\]

**STCF**
• **Round Robin**
  - Run queue is treated as a circular FIFO queue
  - Each job is given a time slice (or scheduling quantum)
    - Multiple of the timer-interrupt period or the timer *tick*
    - Too short $\rightarrow$ higher context switch overhead
    - Too long $\rightarrow$ less responsive
    - Usually 10 ~ 100ms
  - Runs a job for a time slice and then switches to the next job in the run queue
  - Preemptive
  - No starvation
  - Improved response time $\rightarrow$ great for time-sharing
SJF vs. RR

- RR focuses on a new metric: “response time”
  \[ T_{response} = T_{firstrun} - T_{arrival} \]
  
  - Typically, RR has higher turnaround time than SJF, but better response time

\[ T_{response} = (0 + 30 + 60)/3 = 30 \]

\[ T_{response} = (0 + 10 + 20)/3 = 10 \]

\[ T_{turnaround} = (30 + 60 + 90)/3 = 60 \]

\[ T_{turnaround} = (70 + 80 + 90)/3 = 80 \]

A(30), B(30), C(30)

SJF

RR
(Static) Priority Scheduling

• Each job has a (static) priority
  – cf.) nice(), renice(), setpriority(), getpriority()

• Choose the job with the highest priority to run next
  – ex.) shortest job in SJF

• Round-robin or FIFO within the same priority

• Can be either preemptive or non-preemptive

• Starvation problem
  – If there is an endless supply of high priority jobs, no low priority job will ever run
Incorporating I/O

- **I/O-aware scheduling**
  - Assumption 4 relaxed
  - Overlap computation with I/O
  - Treat each CPU burst as an independent job

- **Example:** A (interactive) + B (CPU-intensive)
Towards a General CPU Scheduler

• Goals
  – Optimize turnaround time
  – Minimize response time for interactive jobs

• Challenge: No \textit{a priori} knowledge on the workloads
  – The run time of each job is known (Assumption 5)

• How can the scheduler learn the characteristics of the jobs and make better decisions?
  – Learn from the past to predict the future (as in branch predictors or cache algorithms)
MLFQ

- Multi-Level Feedback Queue
  - A number of distinct queues for each priority level
  - Priority scheduling b/w queues, round-robin in the same queue

**Rule 1:** If Priority(A) > Priority(B), A runs (B doesn’t).
**Rule 2:** If Priority(A) = Priority(B), A & B run in RR.

- Priority is varied based on its observed behavior

```
Queue headers
Priority 4 ───┘
Priority 3 ───┘
Priority 2 ───┘
Priority 1 ───┘

Runnable processes
```
Changing Priority

• **Typical workload: a mix of**
  – Interactive jobs: short-running, require fast response time
  – CPU-intensive jobs: need a lot of CPU time, don’t care about response time

• **Attempt #1: Dynamic Priority Change**

**Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost queue).

**Rule 4a:** If a job uses up an entire time slice while running, its priority is reduced (i.e. moves down one queue).

**Rule 4b:** If a job gives up the CPU before the time slice is up, it stays at the same priority level.
Scheduling Under Rules 1-4

- **Workload**
  - A: long-running job, B: short-running job, C: interactive job
Priority Boost

• Problems in Attempt #1
  – Long-running jobs can starve due to too many interactive jobs
  – A malicious user can game the scheduler by relinquishing the CPU just before the time slice is expired
  – A program may change its behavior over time

• Attempt #2: Priority Boost

**Rule 5:** After some time period $S$, move all the jobs in the system to the topmost queue.
Scheduling Under Rules 1-5

Without Priority Boost

With Priority Boost

... starvation ...

EEE3052: Introduction to Operating Systems, Fall 2017, Jinkyu Jeong (jinkyu@skku.edu)
Better Accounting

• Attempt #3: Revise Rule 4a/4b for better accounting

Rule 4: Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced.
UNIX Scheduler

• **MLFQ**
  – Preemptive priority scheduling
  – Time-shared based on time slice
  – Processes dynamically change priority
  – 3~4 classes spanning ~170 priority levels (Solaris 2)

• Favor interactive processes over CPU-bound processes

• Use aging: no starvation
  – Increase priority as a function of wait time
  – Decrease priority as a function of CPU time

• Many ugly heuristics for voo-doo constants