Announcement

- Homework 2 due on October 26th
- Project 1 due on October 27th
Scheduling Overview

- In discussing process management and synchronization, we talked about context switching among processes/threads on the ready queue.
- But we have glossed over the details of exactly which thread is chosen from the ready queue.
- Making this decision is called **scheduling**.
- In this lecture, we’ll look at:
  - The goals of scheduling
  - Starvation
  - Various well-known scheduling algorithms
  - Standard Unix scheduling algorithm
Multiprogramming

- In a multiprogramming system, we try to increase CPU utilization and job throughput by overlapping I/O and CPU activities
  - Doing this requires a combination of mechanisms and policy
- We have covered the mechanisms
  - Context switching, how and when it happens
  - Process queues and process states
- Now we’ll look at the policies
  - Which process (thread) to run, for how long, etc.
- We’ll refer to schedulable entities as jobs (standard usage) – could be processes, threads, etc.
Scheduling

- Deciding which process/thread should occupy the resource (CPU, disk, etc)

I want to play it

Whose turn is it?

Process 1

Process 2

Process 3
When does OS need to do schedule?

- A new process starts
- A running process exits
- A running process is blocked
- I/O interrupt (some processes will be ready)
- Clock interrupt (every 10 milliseconds)
Preemptive vs. Non-preemptive

- **Non-preemptive scheduling:**
  - The running process keeps the CPU until it voluntarily gives up the CPU
    - process exits
    - switches to blocked state
    - 1 and 4 only (no 3)

- **Preemptive scheduling:**
  - The running process can be interrupted and must release the CPU (can be forced to give up CPU)
What are the scheduling objectives?

- Anyone?

CPU

I want to play

Whose turn is it?

Process 1

Process 2

Process 3

10/21/17
CSE 120 – Scheduling and Deadlock
Scheduling Objectives

- Fair (nobody cries)
- Priority (lady first)
- Efficiency (make best use of equipment)
- Encourage good behavior (good boy/girl)
- Support heavy loads (degrade gracefully)
- Adapt to different environments (interactive, real-time, multi-media)
Performance Criteria

- Fairness
- **Efficiency**: keep resources as busy as possible
- **Throughput**: # of processes that completes in unit time
- **Turnaround Time** (also called elapse time)
  - amount of time to execute a particular process from the time its entered
- **Waiting Time**
  - amount of time process has been waiting in ready queue
- **Response Time**
  - amount of time from when a request was first submitted until first response is produced.
  - predictability and variance
- **Meeting Deadlines**: avoid losing data
Different Systems, Different Focuses

- For all
  - Fairness, policy enforcement, resource balance

- Batch Systems
  - Max throughput, min turnaround time, max CPU utilization

- Interactive Systems
  - Min Response time, best proportionality

- Real-Time Systems
  - Predictability, meeting deadlines
Program Behaviors Considered in Scheduling

- Is it I/O bound? Example?
- Is it CPU bound? Example?
- Batch or interactive environment
- Urgency
- Priority
- Frequency of page faults
- Frequency of preemption
- How much execution time it has already received
- How much execution time it needs to complete
Single Processor Scheduling Algorithms

- **Batch systems**
  - First Come First Serve (FCFS)
  - Short Job First

- **Interactive Systems**
  - Round Robin
  - Priority Scheduling
  - Multi Queue & Multi-level Feedback
  - Shortest process time
  - Guaranteed Scheduling
  - Lottery Scheduling
  - Fair Sharing Scheduling
First Come First Serve (FCFS)

- Process that requests the CPU FIRST is allocated the CPU FIRST.
  - Also called FIFO
- Non-preemptive
- Used in Batch Systems
- Real life analogy: Fast food restaurant
- Implementation: FIFO queues
  - A new process enters the tail of the queue
  - The schedule selects from the head of the queue.
- Performance Metric: Average Waiting Time.
- Given Parameters:
  - Duration(in ms), Arrival Time and Order
The final schedule:

<table>
<thead>
<tr>
<th>Process</th>
<th>Duration</th>
<th>Order</th>
<th>Arrival Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>24</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>4</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

P1 waiting time: 0
P2 waiting time: 24
P3 waiting time: 27

The average waiting time:
\[
\frac{(0+24+27)}{3} = 17
\]
Problems with FCFS

- Non-preemptive
- Not optimal AWT
  - Average waiting time can be large if small jobs wait behind long ones
    - You have a basket, but you’re stuck behind someone with a full shopping cart at Von’s.
- Solution?
  - Express lane (12 items or less)
Shortest Job First (SJF)

- Schedule the job with the shortest elapse time (duration) first
- Scheduling used in Batch Systems
- Two types: Non-preemptive & Preemptive
- Requirement: the elapse time needs to know in advance
- Optimal if all the jobs are available simultaneously (provable).
  - Gives the best possible AWT (average waiting time)
- Real life analogy?
  - Express lane in supermarket
  - Shortest important task first

--The 7 Habits of Highly Effective People
Non-preemptive SJF: Example

<table>
<thead>
<tr>
<th>Process</th>
<th>Duration</th>
<th>Order</th>
<th>Arrival Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>8</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>7</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>P4</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

P4 waiting time: 0
P1 waiting time: 3
P3 waiting time: 9
P2 waiting time: 16

The total time is: 24
The average waiting time (AWT): 
\[
\frac{0+3+9+16}{4} = 7
\]
# Comparing to FCFS

<table>
<thead>
<tr>
<th>Process</th>
<th>Duration</th>
<th>Order</th>
<th>Arrival Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>8</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>7</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>P4</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

P1 waiting time: 0  
P2 waiting time: 6  
P3 waiting time: 14  
P4 waiting time: 21

- The total time is the same (why?)
- The average waiting time (AWT): \( \frac{0+6+14+21}{4} = 10.25 \) (comparing to 7)

10/21/17
SJF is not always optimal

- Is SJF optimal if all the jobs are not available simultaneously?

<table>
<thead>
<tr>
<th>Process</th>
<th>Duration</th>
<th>Order</th>
<th>Arrival Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>10</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

P1 waiting time: 0
P2 waiting time: 8

The average waiting time (AWT): 
\[
\frac{(0+8)}{2} = 4
\]
Preemptive SJF

- Also called **Shortest Remaining Time First**
  - Schedule the job with the shortest remaining time required to complete
- Requirement: the elapse time needs to be known in advance
Preemptive SJF: Same Example

<table>
<thead>
<tr>
<th>Process</th>
<th>Duration</th>
<th>Order</th>
<th>Arrival Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>10</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

P1 waiting time: 4-2 = 2
P2 waiting time: 0

The average waiting time (AWT):
(0+2)/2 = 1

No CPU waste!!!
A Problem with SJF

- **Starvation**
  - In some condition, a job is waiting for ever
  - Example: SJF
    - Process A with elapse time of 1 hour arrives at time 0
    - But every 1 minute, a short process with elapse time of 2 minutes arrive
    - Result of SJF: A never gets to run
Interactive Scheduling Algorithms

- Usually preemptive
  - Time is sliced into quantum (time intervals)
  - Scheduling decision is also made at the beginning of each quantum

- Performance Criteria
  - Min Response time
  - best proportionality

- Representative algorithms:
  - Priority-based
  - Round-robin
  - Multi Queue & Multi-level Feedback
  - Shortest process time
  - Guaranteed Scheduling
  - Lottery Scheduling
  - Fair Sharing Scheduling
Priority Scheduling

- Each job is assigned a priority.
- FCFS within each priority level.
- Select highest priority job over lower ones.
- Rational: higher priority jobs are more mission-critical
  - Example: DVD movie player vs. send email

- Real life analogy?
  - Boarding at airports

- Problems:
  - May not give the best AWT
  - indefinite blocking or starvation a process
Set Priority

- Two approaches
  - Static (for system with well known and regular application behaviors)
  - Dynamic (otherwise)

- Priority may be based on:
  - Cost to user.
  - Importance of user.
  - Aging
  - Percentage of CPU time used in last X hours.
### Priority Scheduling: Example

<table>
<thead>
<tr>
<th>Process</th>
<th>Duration</th>
<th>Priority</th>
<th>Arrival Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>6</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>8</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>7</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>P4</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

- **P2** waiting time: 0
- **P4** waiting time: 8
- **P3** waiting time: 11
- **P1** waiting time: 18

The average waiting time (AWT): 
\[
\frac{(0+8+11+18)}{4} = 9.25
\]
(worse than SJF)
Priority in Unix

```
ps -l

  FS  UID  PID  PPID  C  PRI  NI ADDR  SZ WCHAN  TTY  TIME CMD
100  S  14828  2047  2045  0   79   4  -  822 rt_sig pts/1  00:00:00 csh
  00  R  14828  23001  2047  0   80   4  -  791     pts/l  00:00:00 ps
```

yyzhou@csil-linuxl:~]$
Be “nice” in Unix

```
NICE(1)  FSF  NICE(1)

NAME
nice - run a program with modified scheduling priority

SYNOPSIS
nice [OPTION] [COMMAND [ARG]...]

DESCRIPTION
Run COMMAND with an adjusted scheduling priority. With no COMMAND, print the current scheduling priority. ADJUST is 10 by default. Range goes from -20 (highest priority) to 19 (lowest).

-ADJUST
   increment priority by ADJUST first

-n, --adjustment=ADJUST
   same as -ADJUST

--help display this help and exit

--version
```

10/21/17 CSE 120 – Scheduling and Deadlock
For real-time (predictable) systems, priority is often used to isolate a process from those with lower priority. *Priority inversion* is a risk unless all resources are jointly scheduled.

- A solution: priority inheritance

How can this be avoided?
Round-robin

- One of the oldest, simplest, most commonly used scheduling algorithm
- Select process/thread from ready queue in a round-robin fashion (take turns)
- Real life analogy?

Problem:
- Do not consider priority
- Context switch overhead
# Round-robin: Example

<table>
<thead>
<tr>
<th>Process</th>
<th>Duration</th>
<th>Order</th>
<th>Arrival Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Suppose time quantum is: 1 unit, P1, P2 & P3 never block

P1 waiting time: 4  
P2 waiting time: 6  
P3 waiting time: 6  

The average waiting time (AWT):  
\[(4+6+6)/3 = 5.33\]
Time Quantum

- Time slice too large
  - FIFO behavior
  - Poor response time
- Time slice too small
  - Too many context switches (overheads)
  - Inefficient CPU utilization
- Heuristic: 70-80% of jobs block within time-slice
- Typical time-slice 10 to 100 ms
- Time spent in system depends on size of job.
Combining Algorithms

- Scheduling algorithms can be combined
  - Have multiple queues
  - Use a different algorithm for each queue
  - Move processes among queues

- Example: Multiple-level feedback queues (MLFQ)
  - Multiple queues representing different job types
    - Interactive, CPU-bound, batch, system, etc.
  - Queues have priorities, jobs on same queue scheduled RR
  - Jobs can move among queues based upon execution history
    - Feedback: Switch from interactive to CPU-bound behavior
Multi-level Feedback Algorithm: Example
Unix Scheduler

- The canonical Unix scheduler uses a MLFQ
  - 3-4 classes spanning ~170 priority levels (the higher the better)
    - Timesharing: first 60 priorities
    - System: next 40 priorities
    - Real-time: next 60 priorities
    - Interrupt: next 10 (Solaris)

- Priority scheduling across queues, RR within a queue
  - The process with the highest priority always runs
  - Processes with the same priority are scheduled RR

- Processes dynamically change priority
  - Increases over time if process blocks before end of quantum
  - Decreases over time if process uses entire quantum
Motivation of Unix Scheduler

- The idea behind the Unix scheduler is to reward interactive processes over CPU hogs.
- Interactive processes (shell, editor, etc.) typically run using short CPU bursts:
  - They do not finish quantum before waiting for more input.
- Want to minimize response time:
  - Time from keystroke (putting process on ready queue) to executing keystroke handler (process running).
  - Don’t want editor to wait until CPU hog finishes quantum.
- This policy delays execution of CPU-bound jobs:
  - But that’s ok.
Scheduling Summary

- Scheduler (dispatcher) is the module that gets invoked when a context switch needs to happen.
- Scheduling algorithm determines which process runs, where processes are placed on queues.
- Many potential goals of scheduling algorithms:
  - Utilization, throughput, wait time, response time, etc.
- Various algorithms to meet these goals:
  - FCFS/FIFO, SJF, Priority, RR
- Can combine algorithms:
  - Multiple-level feedback queues
  - Unix example