Testing Project 2

```java
threadA() {
    create threadB
    lock
    unlock
}

threadB() {
    lock
    unlock
}
```

October 8, 2015

EECS 482 – Lecture 10

---

**Deadlock**

- Over-constrained synchronization:
  - Cycle of threads holding some resources and waiting on some resources held by others

```
Thread A
  x.lock
  y.lock
  ...
  y.unlock
  x.unlock

Thread B
  y.lock
  x.lock
  ...
  x.unlock
  y.unlock
```

October 8, 2015

EECS 482 – Lecture 10

---

**Dining philosophers**

- 5 philosophers sit at round table
- 1 chopstick between each pair of philosophers
- Each philosopher needs 2 chopsticks to eat
Strategies for handling deadlock

- Ignore
- Detect and fix
  - Detect cycles in the wait-for graph
  - How to fix once detected?
    - Grab resources
    - Roll back execution
- Prevent

Four necessary conditions for deadlock

- Limited resources
  - Not enough to serve all threads simultaneously
- No preemption
  - Can’t force threads to give up resources
- Hold and wait
  - Threads hold resources while waiting to acquire other resources
- Cyclical chain of requests

Eliminating circular chain

- Impose global ordering of resources

Global ordering of resources

Preventing deadlock

- What if we don’t grant resources that will lead to cycle in waits-for-graph?

CPU scheduling

- How to choose next thread to run?
- What are the goals of a CPU scheduler?
Maximize Performance

- Minimize average response time
  - Elapsed time to do each job
- Maximize throughput of entire system
  - Rate at which jobs complete in the system

Response Time

Throughput

Fairness

- Share CPU among threads in equitable manner
- How to share between 1 big and 1 small job?
  - Response time proportional to job size?
- Fairness often conflicts with response time

Starvation is a non-goal

- We have seen that starvation can be outcome of synchronization
  - Example: Readers can starve writers
- Starvation can also be outcome of scheduling
  - Example: Low priority process is starved if constant stream of high priority processes

First-come, first-served (FCFS)

- FIFO ordering between jobs
- No preemptions
  - Thread runs until it calls yield() or blocks
  - No timer interrupts
- Pro: Simple
- Cons?
  - Short jobs can get stuck behind long jobs
  - Not interactive

FCFS Example

- Job A: Arrives at t=0, takes 100 seconds
- Job B: Arrives at t=0+, takes 1 second

A
B

1 = 0

A’s response time = 100
B’s response time = 101
Average response time = 100.5

Round Robin

- Improve average response time for short jobs
- Periodically preempt all jobs (mainly long-running ones)
  - Every job gets a fixed time slice on CPU before being preempted
- How to implement this?
  - Use timer interrupts
**Round Robin Example**

- Job A: Arrives at $t=0$, takes 100 seconds
- Job B: Arrives at $t=0^+$, takes 1 second

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td></td>
<td>A</td>
</tr>
</tbody>
</table>

- A's response time = 101
- B's response time = 2
- Average response time = 51.5

**Round Robin**

- **Pro:** Good for interactive computing
- **Cons?**
  - More context-switching overhead
- **How to choose time slice?**
  - Big time slice: degenerates to FCFS
  - Small time slice: more context switching overhead
  - Typically a compromise, e.g., 10 ms (if each context switch takes 0.1 ms, this leads to 1% overhead)
- **Does RR always reduce response time vs. FCFS?**

**Round Robin vs. FCFS**

- Jobs A and B arrive at $t=0$, both take 100 secs

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>199</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>200</td>
<td></td>
<td>A</td>
</tr>
</tbody>
</table>

- Average response time with FCFS = 150
- Average response time with RR = 199.5

**STCF (shortest time to completion first)**

- Run whichever job has least amount of work to do before finishing or blocking
  - Preempt current job if shorter job arrives
- **Finish short jobs first**
  - Improves response time of short jobs (by a lot)
  - Hurts response time of long jobs (by a little)
- **STCF gives optimal average response time**

**Analysis of STCF**

- A's response time increases by B's runtime
- B's response time decreases by A's runtime
- Since A's runtime > B's runtime, average response time decreases

**STCF**

- **Pro:** Optimal average response time
- **Cons?**
  - Potential starvation for long jobs
  - Needs knowledge of future
- **How to estimate the time a job will run for?**
  - Ask the job or the user?
  - Use past to predict future