Review: Deadlock

- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
  - Deadlock: circular waiting for resources
  - Deadlock $\Rightarrow$ Starvation, but not other way around

- Four conditions for deadlocks
  - Mutual exclusion
    - Only one thread at a time can use a resource
  - Hold and wait
    - Thread holding at least one resource is waiting to acquire additional resources held by other threads
  - No preemption
    - Resources are released only voluntarily by the threads
  - Circular wait
    - There exists a set $\{T_1, ..., T_n\}$ of threads with a cyclic waiting pattern

Review: Resource Allocation Graph Examples

- Recall:
  - request edge - directed edge $T_i \rightarrow R_j$
  - assignment edge - directed edge $R_j \rightarrow T_i$

Review: Methods for Handling Deadlocks

- Allow system to enter deadlock and then recover
  - Requires deadlock detection algorithm
  - Some technique for selectively preempting resources and/or terminating tasks
- Ensure that system will never enter a deadlock
  - Need to monitor all lock acquisitions
  - Selectively deny those that might lead to deadlock
- Ignore the problem and pretend that deadlocks never occur in the system
  - used by most operating systems, including UNIX
### Goals for Today

- Preventing Deadlock
- Scheduling Policy goals
- Policy Options
- Implementation Considerations

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

### Deadlock Detection Algorithm

- Only one of each type of resource ⇒ look for loops
- More General Deadlock Detection Algorithm
  - Let [X] represent an m-ary vector of non-negative integers (quantities of resources of each type):
    - [FreeResources]: Current free resources each type
    - [Request]<sub>X</sub>: Current requests from thread X
    - [Alloc]<sub>X</sub>: Current resources held by thread X
  - See if tasks can eventually terminate on their own
    - [Avail] = [FreeResources]
    - Add all nodes to UNFINISHED
    - do {
      - done = true
      - Foreach node in UNFINISHED {
        - if ([Request]<sub>node</sub> <= [Avail]) {
          - remove node from UNFINISHED
          - [Avail] = [Avail] + [Alloc]<sub>node</sub>
          - done = false
        }
      } until(done)
    - Nodes left in UNFINISHED ⇒ deadlocked

### Deadlock Detection Algorithm Example

- \[[Available] = [0,0]\]
- \[[Available] = [1,0]\]
- \[[Available] = [1,1]\]

### What to do when detect deadlock?

- Terminate thread, force it to give up resources
  - In Bridge example, Godzilla picks up a car, hurls it into the river. Deadlock solved!
  - Shoot a dining philosopher
  - But, not always possible - killing a thread holding a mutex leaves world inconsistent
- Preempt resources without killing off thread
  - Take away resources from thread temporarily
  - Doesn’t always fit with semantics of computation
- Roll back actions of deadlocked threads
  - Hit the rewind button on TiVo, pretend last few minutes never happened
  - For bridge example, make one car roll backwards (may require others behind him)
  - Common technique in databases (transactions)
  - Of course, if you restart in exactly the same way, may reenter deadlock once again
Techniques for Preventing Deadlock

• **Infinite resources**
  - Include enough resources so that no one ever runs out of resources. Doesn’t have to be infinite, just large.
  - Give illusion of infinite resources (e.g., virtual memory)
  - Examples:
    » Bay bridge with 12,000 lanes. Never wait!
    » Infinite disk space (not realistic yet?)

• **No Sharing of resources (totally independent threads)**
  - Not very realistic

• **Don’t allow waiting**
  - How the phone company avoids deadlock
    » Call to your Mom in Toledo, works its way through the phone lines, but if blocked get busy signal.
  - Technique used in Ethernet/some multiprocessor nets
    » Everyone speaks at once. On collision, back off and retry

Review: Train Example (Wormhole-Routed Network)

• **Circular dependency (Deadlock!)**
  - Each train wants to turn right
  - Blocked by other trains
  - Similar problem to multiprocessor networks

• **Fix?** Imagine grid extends in all four directions
  - **Force ordering of channels (tracks)**
    » Protocol: Always go east-west first, then north-south
  - Called “dimension ordering” (X then Y)

Banker’s Algorithm for Preventing Deadlock

• **Toward right idea:**
  - State maximum resource needs in advance
  - Allow particular thread to proceed if:
    (available resources - #requested) ≥ max remaining that might be needed by any thread

• **Banker’s algorithm (less conservative):**
  - Allocate resources dynamically
    » Evaluate each request and grant if some ordering of threads is still deadlock free afterward
    » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting
      (\([\text{Max}_{\text{node}}]-[\text{Alloc}_{\text{node}}]\) ≤ [Avail]) for (\([\text{Request}_{\text{node}}]\) ≤ [Avail])
  - Grant request if result is deadlock free (conservative!)
  - Keeps system in a “SAFE” state, i.e., there exists a sequence \(\{T_1, T_2, \ldots, T_n\}\) with \(T_1\) requesting all remaining resources, finishing, then \(T_2\) requesting all remaining resources, etc..

  - Algorithm allows the sum of maximum resource needs of all current threads to be greater than total resources
Banker’s Algorithm Example

- Banker’s algorithm with dining philosophers
  - “Safe” (won’t cause deadlock) if when try to grab chopstick either:
    » Not last chopstick
    » Is last chopstick but someone will have two afterwards
- What if k-handed philosophers? Don’t allow if:
  » It’s the last one, no one would have k
  » It’s 2nd to last, and no one would have k-1
  » It’s 3rd to last, and no one would have k-2

Administrivia

- Project 1 code due this Monday (2/22)
- Autograder will be available by tomorrow morning
- Midterm coming up in two 1/2 weeks
  - Tuesday, 3/9, 3:30 – 6:30 (Requested this room)
  - Should be 2 hour exam with extra time
  - Closed book, one page of hand-written notes (both sides)

Scheduling Assumptions

- CPU scheduling big area of research in early 70’s
- Many implicit assumptions for CPU scheduling:
  » One program per user
  » One thread per program
  » Programs are independent
- Clearly, these are unrealistic but they simplify the problem so it can be solved
  - For instance: is “fair” about fairness among users or programs?
    » If I run one compilation job and you run five, you get five times as much CPU on many operating systems
- The high-level goal: Dole out CPU time to optimize some desired parameters of system

Earlier, we talked about the life-cycle of a thread
- Active threads work their way from Ready queue to Running to various waiting queues.
- Question: How is the OS to decide which of several threads to take off a queue?
  - Obvious queue to worry about is ready queue
  - Others can be scheduled as well, however
- Scheduling: deciding which threads are given access to resources from moment to moment
Assumption: CPU Bursts

- Execution model: programs alternate between bursts of CPU and I/O
  - Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
  - Each scheduling decision is about which job to give to the CPU for use by its next CPU burst
  - With timeslicing, thread may be forced to give up CPU before finishing current CPU burst

Scheduling Policy Goals/Criteria

- Minimize Response Time
  - Minimize elapsed time to do an operation (or job)
  - Response time is what the user sees:
    » Time to echo a keystroke in editor
    » Time to compile a program
  - Real-time Tasks: Must meet deadlines imposed by World
- Maximize Throughput
  - Maximize operations (or jobs) per second
  - Throughput related to response time, but not identical:
    » Minimizing response time will lead to more context switching than if you only maximized throughput
  - Two parts to maximizing throughput
    » Minimize overhead (for example, context-switching)
    » Efficient use of resources (CPU, disk, memory, etc)
- Fairness
  - Share CPU among users in some equitable way
  - Fairness is not minimizing average response time:
    » Better average response time by making system less fair

First-Come, First-Served (FCFS) Scheduling

- First-Come, First-Served (FCFS)
  - Also “First In, First Out” (FIFO) or “Run until done”
    » In early systems, FCFS meant one program scheduled until done (including I/O)
    » Now, means keep CPU until thread blocks
- Example:

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>24</td>
</tr>
<tr>
<td>P₂</td>
<td>3</td>
</tr>
<tr>
<td>P₃</td>
<td>3</td>
</tr>
</tbody>
</table>

Suppose processes arrive in the order: P₁, P₂, P₃
The Gantt chart for the schedule is:

- Waiting time for P₁ = 0; P₂ = 24; P₃ = 27
- Average waiting time: \((0 + 24 + 27)/3 = 17\)
- Average Completion time: \((24 + 27 + 30)/3 = 27\)
- Convoy effect: short process behind long process

FCFS Scheduling (Cont.)

- Example continued:
  - Suppose that processes arrive in order: P₂, P₃, P₁
  - Now, the Gantt chart for the schedule is:

<table>
<thead>
<tr>
<th>P₂</th>
<th>P₃</th>
<th>P₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>30</td>
</tr>
</tbody>
</table>

- Waiting time for P₁ = 6; P₂ = 0; P₃ = 3
- Average waiting time: \((6 + 0 + 3)/3 = 3\)
- Average Completion time: \((3 + 6 + 30)/3 = 13\)

- In second case:
  - average waiting time is much better (before it was 17)
  - Average completion time is better (before it was 27)
- FIFO Pros and Cons:
  - Simple (+)
  - Short jobs get stuck behind long ones (-)
    » Safeway: Getting milk, always stuck behind cart full of small items
Round Robin (RR)

- FCFS Scheme: Potentially bad for short jobs!
  - Depends on submit order
  - If you are first in line at supermarket with milk, you
    don’t care who is behind you, on the other hand...
- Round Robin Scheme
  - Each process gets a small unit of CPU time
    (time quantum), usually 10-100 milliseconds
  - After quantum expires, the process is preempted
    and added to the end of the ready queue.
  - n processes in ready queue and time quantum is q ⇒
    Each process gets 1/n of the CPU time
    In chunks of at most q time units
    No process waits more than (n-1)q time units

• Performance
  - q large ⇒ FCFS
  - q small ⇒ Interleaved (really small ⇒ hyperthreading?)
  - q must be large with respect to context switch,
    otherwise overhead is too high (all overhead)

Example of RR with Time Quantum = 20

- Example:
  - Process
  | Burst Time |
  | P1 | 53 |
  | P2 | 8  |
  | P3 | 68 |
  | P4 | 24 |

- The Gantt chart is:

- Waiting time for P1 = (68-20)+(112-88)=72
  P2 = (20-0)=20
  P3 = (28-0)+(88-48)+(125-108)=85
  P4 = (48-0)+(108-68)=88
- Average waiting time = (72+20+85+88)/4=66
- Average completion time = (125+28+153+112)/4=104

• Thus, Round-Robin Pros and Cons:
  - Better for short jobs, Fair (+)
  - Context-switching time adds up for long jobs (-)

Round-Robin Discussion

- How do you choose time slice?
  - What if too big?
    - Response time suffers
  - What if infinite (∞)?
    - Get back FIFO
  - What if time slice too small?
    - Throughput suffers!
- Actual choices of timeslice:
  - Initially, UNIX timeslice one second:
    - Worked ok when UNIX was used by one or two people.
    - What if three compilations going on? 3 seconds to echo
      each keystroke!
  - In practice, need to balance short-job performance
    and long-job throughput:
    - Typical time slice today is between 10ms - 100ms
    - Typical context-switching overhead is 0.1ms - 1ms
    - Roughly 1% overhead due to context-switching

Comparisons between FCFS and Round Robin

- Assuming zero-cost context-switching time, is RR
  always better than FCFS?
- Simple example: 10 jobs, each take 100s of CPU time
  RR scheduler quantum of 1s
  All jobs start at the same time
- Completion Times:

<table>
<thead>
<tr>
<th>Job #</th>
<th>FIFO</th>
<th>RR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>991</td>
</tr>
<tr>
<td>2</td>
<td>200</td>
<td>992</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>900</td>
<td>999</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>1000</td>
</tr>
</tbody>
</table>

- Both RR and FCFS finish at the same time
- Average response time is much worse under RR!
  - Bad when all jobs same length
- Also: Cache state must be shared between all jobs with
  RR but can be devoted to each job with FIFO
- Total time for RR longer even for zero-cost switch!
Quantum Completion Time Wait Time

Earlier Example with Different Time Quantum

Best FCFS: \[ \begin{array}{c|c|c|c|c}
P_2 & P_3 & P_4 & P_5 \\
8 & 32 & 85 & 8 \\
\end{array} \]

Average

<table>
<thead>
<tr>
<th>Quantum</th>
<th>( P_2 )</th>
<th>( P_3 )</th>
<th>( P_4 )</th>
<th>( P_5 )</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q = 1</td>
<td>84</td>
<td>22</td>
<td>85</td>
<td>57</td>
<td>62</td>
</tr>
<tr>
<td>Q = 5</td>
<td>82</td>
<td>20</td>
<td>85</td>
<td>58</td>
<td>61\frac{1}{2}</td>
</tr>
<tr>
<td>Q = 8</td>
<td>80</td>
<td>8</td>
<td>85</td>
<td>56</td>
<td>57\frac{1}{2}</td>
</tr>
<tr>
<td>Q = 10</td>
<td>82</td>
<td>10</td>
<td>85</td>
<td>68</td>
<td>61\frac{1}{2}</td>
</tr>
<tr>
<td>Q = 20</td>
<td>72</td>
<td>20</td>
<td>85</td>
<td>88</td>
<td>66\frac{1}{4}</td>
</tr>
<tr>
<td>Worst FCFS</td>
<td>68</td>
<td>145</td>
<td>0</td>
<td>121</td>
<td>83\frac{1}{4}</td>
</tr>
</tbody>
</table>

Completion Time

Best FCFS: \[ \begin{array}{c|c|c|c|c}
P_2 & P_3 & P_4 & P_5 \\
85 & 8 & 153 & 32 \\
\end{array} \]

Average

<table>
<thead>
<tr>
<th>Quantum</th>
<th>( P_2 )</th>
<th>( P_3 )</th>
<th>( P_4 )</th>
<th>( P_5 )</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q = 1</td>
<td>137</td>
<td>30</td>
<td>153</td>
<td>81</td>
<td>100\frac{1}{4}</td>
</tr>
<tr>
<td>Q = 5</td>
<td>135</td>
<td>28</td>
<td>153</td>
<td>82</td>
<td>99\frac{1}{4}</td>
</tr>
<tr>
<td>Q = 8</td>
<td>133</td>
<td>16</td>
<td>153</td>
<td>80</td>
<td>95\frac{1}{2}</td>
</tr>
<tr>
<td>Q = 10</td>
<td>135</td>
<td>18</td>
<td>153</td>
<td>92</td>
<td>99\frac{1}{2}</td>
</tr>
<tr>
<td>Q = 20</td>
<td>125</td>
<td>28</td>
<td>153</td>
<td>112</td>
<td>104\frac{1}{4}</td>
</tr>
<tr>
<td>Worst FCFS</td>
<td>121</td>
<td>153</td>
<td>68</td>
<td>145</td>
<td>121\frac{1}{4}</td>
</tr>
</tbody>
</table>

What if we Knew the Future?

- Could we always mirror best FCFS?
- Shortest Job First (SJF):
  - Run whatever job has the least amount of computation to do
  - Sometimes called “Shortest Time to Completion First” (STCF)
- Shortest Remaining Time First (SRTF):
  - Preemptive version of SJF: if job arrives and has a shorter time to completion than the remaining time on the current job, immediately preempt CPU
  - Sometimes called “Shortest Remaining Time to Completion First” (SRTCF)
- These can be applied either to a whole program or the current CPU burst of each program
  - Idea is to get short jobs out of the system
  - Big effect on short jobs, only small effect on long ones
  - Result is better average response time

Discussion

- SJF/SRTF are the best you can do at minimizing average response time
  - Provably optimal (SJF among non-preemptive, SRTF among preemptive)
  - Since SRTF is always at least as good as SJF, focus on SRTF
- Comparison of SRTF with FCFS and RR
  - What if all jobs the same length?
    - SRTF becomes the same as FCFS (i.e. FCFS is best can do if all jobs the same length)
  - What if jobs have varying length?
    - SRTF (and RR): short jobs not stuck behind long ones

Example to illustrate benefits of SRTF

A or B

| C |
| Cs | Cs | Cs |
| I/O | I/O | I/O |

- Three jobs:
  - A, B: both CPU bound, run for week
  - C: I/O bound, loop 1ms CPU, 9ms disk I/O
  - If only one at a time, C uses 90% of the disk, A or B could use 100% of the CPU
- With FIFO:
  - Once A or B get in, keep CPU for two weeks
- What about RR or SRTF?
  - Easier to see with a timeline
SRTF Example continued:

Disk Utilization: 9/201 ~ 4.5%

RR 100ms time slice

Disk Utilization: ~90% but lots of wakeups!

RR 1ms time slice

Disk Utilization: 90%

Summary (Deadlock)

- Four conditions required for deadlocks
  - Mutual exclusion
    » Only one thread at a time can use a resource
  - Hold and wait
    » Thread holding at least one resource is waiting to acquire additional resources held by other threads
  - No preemption
    » Resources are released only voluntarily by the threads
  - Circular wait
    » ∃ set \( \{ T_1, \ldots, T_n \} \) of threads with a cyclic waiting pattern

- Deadline detection
  - Attempts to assess whether waiting graph can ever make progress

- Deadline prevention
  - Assess, for each allocation, whether it has the potential to lead to deadlock
  - Banker’s algorithm gives one way to assess this

Summary (Scheduling)

- Scheduling: selecting a waiting process from the ready queue and allocating the CPU to it

- FCFS Scheduling:
  - Run threads to completion in order of submission
  - Pros: Simple
  - Cons: Short jobs get stuck behind long ones

- Round-Robin Scheduling:
  - Give each thread a small amount of CPU time when it executes: cycle between all ready threads
  - Pros: Better for short jobs
  - Cons: Poor when jobs are same length

- Shortest Job First (SJF)/Shortest Remaining Time First (SRTF):
  - Run whatever job has the least amount of computation to do/least remaining amount of computation to do
  - Pros: Optimal (average response time)
  - Cons: Hard to predict future, Unfair

SRTF Further discussion

- Starvation
  - SRTF can lead to starvation if many small jobs!
    - Large jobs never get to run

- Somehow need to predict future
  - How can we do this?
    - Some systems ask the user
      » When you submit a job, have to say how long it will take
      » To stop cheating, system kills job if takes too long
    - But: Even non-malicious users have trouble predicting runtime of their jobs

- Bottom line, can't really know how long job will take
  - However, can use SRTF as a yardstick for measuring other policies
    - Optimal, so can't do any better

- SRTF Pros & Cons
  - Optimal (average response time) (+)
  - Hard to predict future (-)
  - Unfair (-)