CS162
Operating Systems and Systems Programming
Lecture 11

Scheduling (finished), Deadlock, Address Translation

March 1st, 2017
Prof. Ion Stoica
http://cs162.eecs.Berkeley.edu
Recap: What if we Knew the Future?

- Could we always mirror best FCFS?
- Shortest Job First (SJF):
  - Run whatever job has 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 to whole program or current CPU burst
  - 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
Recap: 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

• 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%

Disk Utilization: ~90% but lots of wakeups!

Disk Utilization: 90%

RR 100ms time slice

RR 1ms time slice

SRTF
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: hard to predict job's runtime even for non-malicious users
SRTF Further discussion (Cont.)

- 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 (-)
Predicting the Length of the Next CPU Burst

- **Adaptive**: Changing policy based on past behavior
  - CPU scheduling, in virtual memory, in file systems, etc.
  - Works because programs have predictable behavior
    - If program was I/O bound in past, likely in future
    - If computer behavior were random, wouldn’t help

- **Example**: SRTF with estimated burst length
  - Use an estimator function on previous bursts:
    Let \( t_{n-1}, t_{n-2}, t_{n-3}, \) etc. be previous CPU burst lengths. Estimate next burst \( \tau_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \ldots) \)
  - Function \( f \) could be one of many different time series estimation schemes (Kalman filters, etc)
  - For instance, exponential averaging
    \[
    \tau_n = \alpha t_{n-1} + (1-\alpha) \tau_{n-1}
    \]
    with \( 0 < \alpha \leq 1 \)
Multi-Level Feedback Scheduling

- Another method for exploiting past behavior (first use in CTSS)
  - Multiple queues, each with different priority
    - Higher priority queues often considered “foreground” tasks
  - Each queue has its own scheduling algorithm
    - e.g. foreground – RR, background – FCFS
    - Sometimes multiple RR priorities with quantum increasing exponentially (highest: 1ms, next: 2ms, next: 4ms, etc)
- Adjust each job’s priority as follows (details vary)
  - Job starts in highest priority queue
  - If timeout expires, drop one level
  - If timeout doesn't expire, push up one level (or to top)
Scheduling Details

- Result approximates SRTF:
  - CPU bound jobs drop like a rock
  - Short-running I/O bound jobs stay near top
- Scheduling must be done between the queues
  - Fixed priority scheduling:
    » serve all from highest priority, then next priority, etc.
  - Time slice:
    » each queue gets a certain amount of CPU time
    » e.g., 70% to highest, 20% next, 10% lowest
Scheduling Details

• **Countermeasure**: user action that can foil intent of OS designers
  – For multilevel feedback, put in a bunch of meaningless I/O to keep job’s priority high
  – Of course, if everyone did this, wouldn’t work!

• **Example of Othello program:**
  – Playing against competitor, so key was to do computing at higher priority the competitors.
    » Put in `printf`'s, ran much faster!
Real-Time Scheduling (RTS)

• Efficiency is important but **predictability** is essential:
  – We need to predict with confidence worst case response times for systems
  – In RTS, performance guarantees are:
    » Task- and/or class centric and often ensured a priori
  – In conventional systems, performance is:
    » System/throughput oriented with post-processing (… wait and see …)
  – Real-time is about enforcing predictability, and does not equal fast computing!!!

• Hard Real-Time
  – *Attempt to meet all deadlines*
  – EDF (Earliest Deadline First), LLF (Least Laxity First), RMS (Rate-Monotonic Scheduling), DM (Deadline Monotonic Scheduling)

• Soft Real-Time
  – *Attempt to meet deadlines with high probability*
  – Minimize miss ratio / maximize completion ratio (firm real-time)
  – Important for multimedia applications
  – CBS (Constant Bandwidth Server)
Example: Workload Characteristics

- Tasks are preemptable, independent with arbitrary arrival (=release) times
- Tasks have deadlines (D) and known computation times (C)
- Example Setup:
Example: Round-Robin Scheduling Doesn’t Work

Time

T1

T2

T3

T4

Missed deadline!!
Earliest Deadline First (EDF)

- Tasks periodic with period $P$ and computation $C$ in each period: $(P, C)$
- Preemptive priority-based dynamic scheduling
- Each task is assigned a (current) priority based on how close the absolute deadline is
- The scheduler always schedules the active task with the closest absolute deadline

$T_1 = (4,1)$

$T_2 = (5,2)$

$T_3 = (7,2)$
A Final Word On Scheduling

• When do the details of the scheduling policy and fairness really matter?
  – When there aren’t enough resources to go around

• When should you simply buy a faster computer?
  – (Or network link, or expanded highway, or …)
  – One approach: Buy it when it will pay for itself in improved response time
    » Assuming you’re paying for worse response time in reduced productivity, customer angst, etc…
    » Might think that you should buy a faster X when X is utilized 100%, but usually, response time goes to infinity as utilization $\Rightarrow 100$

• An interesting implication of this curve:
  – Most scheduling algorithms work fine in the “linear” portion of the load curve, fail otherwise
  – Argues for buying a faster X when hit “knee” of curve
Deadlock
• Starvation vs. Deadlock
  – Starvation: thread waits indefinitely
    » Example, low-priority thread waiting for resources constantly in use by high-priority threads
  – Deadlock: circular waiting for resources
    » Thread A owns Res 1 and is waiting for Res 2
    » Thread B owns Res 2 and is waiting for Res 1

– Deadlock ⇒ Starvation but not vice versa
  » Starvation can end (but doesn’t have to)
  » Deadlock can’t end without external intervention
Conditions for Deadlock

- Deadlock not always deterministic – Example 2 mutexes:

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.P();</td>
<td>y.P();</td>
</tr>
<tr>
<td>y.P();</td>
<td>x.P();</td>
</tr>
<tr>
<td>y.V();</td>
<td>x.V();</td>
</tr>
<tr>
<td>x.V();</td>
<td>y.V();</td>
</tr>
</tbody>
</table>

  - Deadlock won’t always happen with this code
    » Have to have exactly the right timing (“wrong” timing?)
    » So you release a piece of software, and you tested it, and there it is, controlling a nuclear power plant…

- Deadlocks occur with multiple resources
  - Means you can’t decompose the problem
  - Can’t solve deadlock for each resource independently

- Example: System with 2 disk drives and two threads
  - Each thread needs 2 disk drives to function
  - Each thread gets one disk and waits for another one
Bridge Crossing Example

• Each segment of road can be viewed as a resource
  – Car must own the segment under them
  – Must acquire segment that they are moving into
• For bridge: must acquire both halves
  – Traffic only in one direction at a time
  – Problem occurs when two cars in opposite directions on bridge: each acquires one segment and needs next
• If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
  – Several cars may have to be backed up
• Starvation is possible
  – East-going traffic really fast ⇒ no one goes west
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)
Dining Philosophers Problem

• Five chopsticks/Five philosophers
  – Free-for all: Philosopher will grab any one they can
  – Need two chopsticks to eat
• What if all grab at same time?
  – Deadlock!
• How to fix deadlock?
  – Make one of them give up a chopstick (Hah!)
  – Eventually everyone will get chance to eat
• How to prevent deadlock?
  – Never let philosopher take last chopstick if no hungry philosopher has two chopsticks afterwards
Four requirements for Deadlock

• 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 thread holding the resource, after thread is finished with it

• Circular wait
  – There exists a set \( \{T_1, \ldots, T_n\} \) of waiting threads
    » \( T_1 \) is waiting for a resource that is held by \( T_2 \)
    » \( T_2 \) is waiting for a resource that is held by \( T_3 \)
    » …
    » \( T_n \) is waiting for a resource that is held by \( T_1 \)
Resource-Allocation Graph

• System Model
  – A set of Threads $T_1, T_2, \ldots, T_n$
  – Resource types $R_1, R_2, \ldots, R_m$
    \(CPU\) cycles, memory space, I/O devices
  – Each resource type $R_i$ has $W_i$ instances
  – Each thread utilizes a resource as follows:
    » Request() / Use() / Release()

• Resource-Allocation Graph:
  – V is partitioned into two types:
    » $T = \{T_1, T_2, \ldots, T_n\}$, the set threads in the system.
    » $R = \{R_1, R_2, \ldots, R_m\}$, the set of resource types in system
  – request edge – directed edge $T_1 \rightarrow R_j$
  – assignment edge – directed edge $R_j \rightarrow T_i$
Resource Allocation Graph Examples

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

• Midterm #1 grades/solutions are available
  – Regrades request deadline: 3/13 at 11:59PM

• Upcoming deadlines:
  – Project 1 final code due on Fri 3/3
  – HW2 due on Mon 3/6
  – Final report for Project 1 due on Mon 3/6
BREAK
Methods for Handling Deadlocks

• Allow system to enter deadlock and then recover
  – Requires deadlock detection algorithm
  – Some technique for forcibly 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
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):
    
    - $[\text{FreeResources}]$: Current free resources each type
    - $[\text{Request}_x]$: Current requests from thread $X$
    - $[\text{Alloc}_x]$: Current resources held by thread $X$
  
  - See if tasks can eventually terminate on their own

$[\text{Avail}] = [\text{FreeResources}]$

Add all nodes to UNFINISHED

do {
  done = true
  Foreach node in UNFINISHED {
    if ($[[\text{Request}_{\text{node}}]] <= [\text{Avail}]$) {
      remove node from UNFINISHED
      $[\text{Avail}] = [\text{Avail}] + [\text{Alloc}_{\text{node}}]$
      done = false
    }
  }
} until(done)

- Nodes left in UNFINISHED ⇒ deadlocked
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 lawyer
  – 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

• Many operating systems use other options
Four requirements for Deadlock

- **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 thread holding the resource, after thread is finished with it.

- **Circular wait**
  - There exists a set \( \{T_1, \ldots, T_n\} \) of waiting threads
    - \( T_1 \) is waiting for a resource that is held by \( T_2 \)
    - \( T_2 \) is waiting for a resource that is held by \( T_3 \)
    - \( \ldots \)
    - \( T_n \) is waiting for a resource that is held by \( T_1 \)
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
  - Inefficient, since have to keep retrying
    » Consider: driving to San Francisco; when hit traffic jam, suddenly you’re transported back home and told to retry!
Techniques for Preventing Deadlock (cont’d)

• Make all threads request everything they’ll need at the beginning.
  – Problem: Predicting future is hard, tend to over-estimate resources
  – Example:
    » If need 2 chopsticks, request both at same time
    » Don’t leave home until we know no one is using any intersection
      between here and where you want to go; only one car on the Bay
      Bridge at a time

• Force all threads to request resources in a particular order
  preventing any cyclic use of resources
  – Thus, preventing deadlock
  – Example (x.P, y.P, z.P,…)
    » Make tasks request disk, then memory, then…
    » Keep from deadlock on freeways around SF by requiring everyone to
      go clockwise
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
      ([Max\text{node}] - [Alloc\text{node}] \leq [Avail]) for ([Request\text{node}] \leq [Avail])
      Grant request if result is deadlock free (conservative!)
Banker’s Algorithm for Preventing Deadlock

• [Avail] = [FreeResources]
  Add all nodes to UNFINISHED
do {
    done = true
    Foreach node in UNFINISHED {
      if ([Request_{node}] <= [Avail]) {
        remove node from UNFINISHED
        [Avail] = [Avail] + [Alloc_{node}]
        done = false
      }
    }
} until(done)

» Technique: pretend each request is granted, then run deadlock detection algorithm, substituting
  
  ([Max_{node}]-[Alloc_{node}] <= [Avail]) for ([Request_{node}] <= [Avail])

Grant request if result is deadlock free (conservative!)
Banker’s Algorithm for Preventing Deadlock

- \([\text{Avail}] = [\text{FreeResources}]\)
  Add all nodes to UNFINISHED
  do {
    done = true
    Foreach node in UNFINISHED {
      if (\([\text{Max}_\text{node}] - [\text{Alloc}_\text{node}] \leq [\text{Avail}]\)) {
        remove node from UNFINISHED
        \([\text{Avail}] = [\text{Avail}] + [\text{Alloc}_\text{node}]\)
        done = false
      }
    }
  }
  until(done)

- Technique: pretend each request is granted, then run deadlock detection algorithm, substituting
  \((\[\text{Max}_\text{node}] - [\text{Alloc}_\text{node}] \leq [\text{Avail}]\)) for \((\[\text{Request}_\text{node}] \leq [\text{Avail}]\))
  Grant request if result is deadlock free (conservative!)
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
      ([Max_node]-[Alloc_node] ≤ [Avail]) for ([Request_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 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 philosopher? Don’t allow if:
    » It’s the last one, no one would have k
    » It’s 2\textsuperscript{nd} to last, and no one would have k-1
    » It’s 3\textsuperscript{rd} to last, and no one would have k-2
    » …
Deadlock Prevention – The Reality

• Deadlock Prevention is HARD
  – How many resources will each thread need?
  – How many total resources are there?

• Also Slow/Impractical
  – Matrix of resources/requirements could be big and dynamic
  – Re-evaluate on every request (even for small/non-contended)
  – Banker’s algorithm assumes everyone asks for max

• REALITY
  – Most OSs don’t bother
  – Programmers job to write deadlock-free programs (e.g. by ordering all resource requests).
Summary

• Starvation (thread waits indefinitely) versus Deadlock (circular waiting for resources)

• 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
    » \( \exists \) set \( \{T_1, \ldots, T_n\} \) of threads with a cyclic waiting pattern

• Techniques for addressing Deadlock
  – Allow system to enter deadlock and then recover
  – Ensure that system will never enter a deadlock
  – Ignore the problem and pretend that deadlocks never occur in system