CS162
Operating Systems and
Systems Programming
Lecture 13

Scheduling 4: Deadlock (Finished)

March 2nd, 2023
Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu
Recall: Linux Completely Fair Scheduler (CFS)

- **Basic Idea:** track CPU time per thread and schedule threads to match up average rate of execution

- **Scheduling Decision:**
  - "Repair" illusion of complete fairness
  - Choose thread with minimum CPU time
  - Closely related to Fair Queueing

- Use a heap-like scheduling queue for this…
  - O(log N) to add/remove threads

- Sleeping threads don’t advance their CPU time, so they get a boost when they wake
  - Get interactivity automatically!

- Differentiation: Use weights! Key Idea: Assign a weight \( w_i \) to each process \( i \) to compute the switching quanta \( Q_i \)
  - Basic equal share: \( Q_i = \text{Target Latency} \cdot \frac{1}{N} \)
  - Weighted Share: \( Q_i = \left( \frac{w_i}{\sum_p w_p} \right) \cdot \text{Target Latency} \)

\[
CFS: \text{Average rate of execution} = \frac{1}{N}
\]
Deadlock: A Deadly type of Starvation

• 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 $\Rightarrow$ Starvation but not vice versa
  – Starvation can end (but doesn’t have to)
  – Deadlock can’t end without external intervention
Example: Single-Lane Bridge Crossing

CA 140 to Yosemite National Park
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

- Deadlock: Shown above when two cars in opposite directions meet in middle
  - Each acquires one segment and needs next
  - Deadlock resolved if one car backs up (preempt resources and rollback)
    » Several cars may have to be backed up

- Starvation (not Deadlock):
  - East-going traffic really fast ⇒ no one gets to go west
Deadlock with Locks

Thread A:  
- x.Acquire();
- y.Acquire();
- ...
- y.Release();
- x.Release();

Thread B:  
- y.Acquire();
- x.Acquire();
- ...
- x.Release();
- y.Release();

• This lock pattern exhibits *non-deterministic deadlock*
  – Sometimes it happens, sometimes it doesn’t!
• This is really hard to debug!
Deadlock with Locks: “Unlucky” Case

Thread A:
x.Acquire();
y.Acquire(); <stalled>
<unreachable>
...
y.Release();
x.Release();

Thread B:
y.Acquire();
x.Acquire(); <stalled>
<unreachable>
...
x.Release();
y.Release();

Neither thread will get to run ⇒ Deadlock
Deadlock with Locks: “Lucky” Case

Thread A:
\[
\begin{align*}
&x.\text{Acquire}(); \\
&y.\text{Acquire}(); \\
&\ldots \\
&y.\text{Release}(); \\
&x.\text{Release}();
\end{align*}
\]

Thread B:
\[
\begin{align*}
&y.\text{Acquire}(); \\
&x.\text{Acquire}(); \\
&\ldots \\
&x.\text{Release}(); \\
&y.\text{Release}();
\end{align*}
\]

Sometimes, schedule won’t trigger deadlock!
Other Types of Deadlock

- Threads often block waiting for resources
  - Locks
  - Terminals
  - Printers
  - CD drives
  - Memory

- Threads often block waiting for other threads
  - Pipes
  - Sockets

- You can deadlock on any of these!
Deadlock with Space

<table>
<thead>
<tr>
<th>Thread A:</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllocateOrWait(1 MB)</td>
<td>AllocateOrWait(1 MB)</td>
</tr>
<tr>
<td>AllocateOrWait(1 MB)</td>
<td>AllocateOrWait(1 MB)</td>
</tr>
<tr>
<td>Free(1 MB)</td>
<td>Free(1 MB)</td>
</tr>
<tr>
<td>Free(1 MB)</td>
<td>Free(1 MB)</td>
</tr>
</tbody>
</table>

If only 2 MB of space, we get same deadlock situation
Dining Lawyers Problem

- Five chopsticks/Five lawyers (really cheap restaurant)
  - Free-for all: Lawyer 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 lawyer take last chopstick if no hungry lawyer has two chopsticks afterwards
  - Can we formalize this requirement somehow?
Four requirements for occurrence of 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 \( T_1 \)
Administrivia

• Welcome to Project 2
  – Please get started earlier than last time!
• Midterm 2
  – Coming up in < 2 weeks! (3/15)
  – Everything up to the midterm is fair game (perhaps deemphasizing the lecture on the day before....)
Detecting Deadlock: 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

- Model:
  - request edge – directed edge $T_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$
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 \(\implies\) deadlocked
How should a system deal with deadlock?

• Four different approaches:
  1. **Deadlock prevention**: write your code in a way that it isn’t prone to deadlock
  2. **Deadlock recovery**: let deadlock happen, and then figure out how to recover from it
  3. **Deadlock avoidance**: dynamically delay resource requests so deadlock doesn’t happen
  4. **Deadlock denial**: ignore the possibility of deadlock

• Modern operating systems:
  – Make sure the *system* isn’t involved in any deadlock
  – Ignore deadlock in applications
    » “Ostrich Algorithm”
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 Mom in Toledo, works way through phone network, 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!
(Virtually) Infinite Resources

- With virtual memory we have “infinite” space so everything will just succeed, thus above example won’t deadlock
  – Of course, it isn’t actually infinite, but certainly larger than 2MB!
Techniques for Preventing Deadlock

• 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.Acquire(), y.Acquire(), z.Acquire(),...)
    » Make tasks request disk, then memory, then...
    » Keep from deadlock on freeways around SF by requiring everyone to go clockwise
Request Resources Atomically (1)

Rather than:

Thread A:
- x.Acquire();
- y.Acquire();
- ...
- y.Release();
- x.Release();

Thread B:
- y.Acquire();
- x.Acquire();
- ...
- x.Release();
- y.Release();

Consider instead:

Thread A:
- Acquire_both(x, y);
- ...
- y.Release();
- x.Release();

Thread B:
- Acquire_both(y, x);
- ...
- x.Release();
- y.Release();
Request Resources Atomically (2)

Or consider this:

Thread A
z.Acquire();
x.Acquire();
y.Acquire();
z.Release();
...
y.Release();
x.Release();

Thread B
z.Acquire();
y.Acquire();
x.Acquire();
z.Release();
...
x.Release();
y.Release();
Acquire Resources in Consistent Order

Rather than:

Thread A:
- x.Acquire();
- y.Acquire();
- ...
- y.Release();
- x.Release();

Thread B:
- y.Acquire();
- x.Acquire();
- ...
- x.Release();
- y.Release();

Consider instead:

Thread A:
- x.Acquire();
- y.Acquire();
- ...
- y.Release();
- x.Release();
- x.Release();

Thread B:
- x.Acquire();
- y.Acquire();
- ...
- x.Release();
- y.Release();

Does it matter in which order the locks are released?
Train Example (Wormhole-Routed Network)

- Circular dependency (Deadlock!)
  - Each train wants to turn right, but is blocked by other trains
- Similar problem to multiprocessor networks
  - Wormhole-Routed Network: Messages trail through network like a “worm”
- 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)
Techniques for Recovering from Deadlock

• Terminate thread, force it to give up resources
  – In Bridge example, Godzilla picks up a car, hurls it into the river. Deadlock solved!
  – Hold dining lawyer in contempt and take away in handcuffs
  – 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
Another view of virtual memory: Pre-empting Resources

Thread A:
AllocateOrWait(1 MB)
AllocateOrWait(1 MB)
Free(1 MB)
Free(1 MB)

Thread B:
AllocateOrWait(1 MB)
AllocateOrWait(1 MB)
Free(1 MB)
Free(1 MB)

• Before: With virtual memory we have “infinite” space so everything will just succeed, thus above example won’t deadlock
  – Of course, it isn’t actually infinite, but certainly larger than 2MB!

• Alternative view: we are “pre-empting” memory when paging out to disk, and giving it back when paging back in
  – This works because thread can’t use memory when paged out
Techniques for Deadlock Avoidance

• Idea: When a thread requests a resource, OS checks if it would result in deadlock
  – If not, it grants the resource right away
  – If so, it waits for other threads to release resources

  **THIS DOES NOT WORK!!!!**

• Example:

  **Thread A:**
  \[
  \text{x.Acquire();}
  \]
  Blocks...
  
  \[
  \text{y.Acquire();}
  \]
  
  \[
  \text{...}
  \]
  
  \[
  \text{y.Release();}
  \]
  
  \[
  \text{x.Release();}
  \]

  **Thread B:**
  \[
  \text{y.Acquire();}
  \]
  
  \[
  \text{x.Acquire();}
  \]
  Wait?
  
  \[
  \text{...}
  \]
  But it’s already too late...
  
  \[
  \text{x.Release();}
  \]
  
  \[
  \text{y.Release();}
  \]
Deadlock Avoidance: Three States

• Safe state
  – System can delay resource acquisition to prevent deadlock

• Unsafe state
  – No deadlock yet…
  – But threads can request resources in a pattern that *unavoidably* leads to deadlock

• Deadlocked state
  – There exists a deadlock in the system
  – Also considered “unsafe”
Deadlock Avoidance

• Idea: When a thread requests a resource, OS checks if it would result in deadlock an unsafe state
  – If not, it grants the resource right away
  – If so, it waits for other threads to release resources

• Example:

Thread A:
- x.Acquire();
- y.Acquire();
- ...
- y.Release();
- x.Release();

Thread B:
- y.Acquire();
- x.Acquire();
- ...
- x.Release();
- y.Release();

Wait until Thread A releases mutex X
Banker’s Algorithm for Avoiding Deadlock

• Toward right idea:
  – State maximum (max) resource needs in advance
  – Allow particular thread to proceed if:
    \[(\text{available resources} - \#\text{requested}) \geq \text{max}\]
    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:
      \[
      ([\text{Max}_{\text{node}}] - [\text{Alloc}_{\text{node}}]) \leq [\text{Avail}] \text{ for } ([\text{Request}_{\text{node}}] \leq [\text{Avail}])
      \]
      Grant request if result is deadlock free (conservative!)
### Banker’s Algorithm for Avoiding Deadlock

- **Toward right idea:**
  - State maximum (max) resource needs in advance
  - Allow particular thread to proceed if: 
    \[
    \text{(available resources - #requested)} \geq \text{max remaining that might be needed by any thread}
    \]

- **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:
      \[
      ([\text{Max}_{\text{node}}]-[\text{Alloc}_{\text{node}}] \leq [\text{Avail}]) \text{ for } ([\text{Request}_{\text{node}}] \leq [\text{Avail}])
      \]
  - Grant request if result is deadlock free (conservative!)

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

- 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:
  
    \[
    ([\text{Max}_{\text{node}}]-[\text{Alloc}_{\text{node}}] \leq [\text{Avail}]) \text{ for } ([\text{Request}_{\text{node}}] \leq [\text{Avail}])
    \]
- Grant request if result is deadlock free (conservative!)
Banker’s Algorithm for Avoiding 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
            [Avail] = [Avail] + [Alloc_{\text{node}}]
            done = false
        }
    }
} until(done)

» 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:
    \( ([\text{Max}_{\text{node}}] - [\text{Alloc}_{\text{node}}] \leq [\text{Avail}]) \) for \( ([\text{Request}_{\text{node}}] \leq [\text{Avail}]) \)
Grant request if result is deadlock free (conservative!)
Banker’s Algorithm for Avoiding Deadlock

• Toward right idea:
  – State maximum (max) 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: 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..
Banker’s Algorithm Example

- Banker’s algorithm with dining lawyers
  - “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 lawyers? 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 Summary

• Four conditions for deadlocks
  – Mutual exclusion
  – Hold and wait
  – No preemption
  – Circular wait

• Techniques for addressing Deadlock
  – **Deadlock prevention:**
    » write your code in a way that it isn’t prone to deadlock
  – **Deadlock recovery:**
    » let deadlock happen, and then figure out how to recover from it
  – **Deadlock avoidance:**
    » dynamically delay resource requests so deadlock doesn’t happen
    » Banker’s Algorithm provides on algorithmic way to do this
  – **Deadlock denial:**
    » ignore the possibility of deadlock