SE350: Operating Systems

Lecture 9: Deadlock
Outline

• Definitions

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

• Techniques for addressing Deadlock
Starvation vs. Deadlock

- **Starvation**: thread waits indefinitely
  - E.g., 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 leads to starvation but not the other way around

- Starvation can end (but doesn’t have to)
- Deadlock can’t end without external intervention
Bridge Crossing Example

• Each segment of road can be viewed as resource
  • Cars must own segment under them and acquire segment they are moving to

• To cross bridge cars 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 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
Conditions for Deadlock

- Deadlock is not always deterministic

<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>

- This code doesn’t always lead to deadlock
  - Must have exactly 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
  - Can’t solve deadlock for each resource independently
Four Requirements for Deadlock

• Mutual exclusion
  • Only limited number of threads at a time can use resource

• Hold and wait
  • Thread hold resources while waiting to acquire additional ones

• No preemption
  • Resources are released only voluntarily by thread holding them

• Circular wait
  • There exists a set $T_1, ..., T_n$ of waiting threads
    • $T_1$ is waiting for resource that is held by $T_2$
    • $T_2$ is waiting for resource that is held by $T_3$
    • ...
    • $T_n$ is waiting for resource that is held by $T_1$
Resource Allocation Graph

• System model
  • 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 $W_{ij}$ instances
  • Each thread utilizes resources as follows
    • Request() / Use() / Release()

• Resource allocation graph
  • $V$ is partitioned into two types
    • $T = \{T_1, \ldots, T_n\}$, set threads in system
    • $R = \{R_1, \ldots, R_m\}$, set of resource types in system
  • Request edge is directed edge $T_i \rightarrow R_j$
  • Assignment edge is directed edge $R_q \rightarrow T_p$
Resource Allocation Graph Examples

1. Simple resource allocation graph
2. Allocation graph with deadlock
3. Allocation graph with cycle, but no deadlock
Each politician needs two chopsticks to eat
Each grabs chopstick on the right first (all right-handed)
Deadlock if all grab chopstick at same time
Deadlock depends on the order of execution
  No deadlock if one was left-handed
Train Example
(Wormhole-Routed Network)

- Each train wants to turn right but is blocked by other trains
- Similar problem to multiprocessor networks
- How to fix this? (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)
Methods for Handling Deadlocks

• Allow system to enter deadlock and then recover
  • Requires deadlock detection algorithm
  • Technique for forcibly preempting resources and/or terminating tasks

• Ensure that system will never enter deadlock
  • Need to monitor all resources acquisitions
  • Selectively deny those that might lead to deadlock

• Ignore problem and pretend deadlocks never occur
  • Used by most operating systems, including UNIX
Deadlock Detection Algorithm

- If there is only one unit of each type of resource ⇒ look for loops
- More general deadlock detection algorithm
  - Let \([x]\) represent \(m\)-ary vector of non-negative integers (units per type)
    - \([\text{FreeResources}]\): Current free resources each type
    - \([\text{Request}_i]\): Current requests from thread \(i\)
    - \([\text{Alloc}_i]\): Current resources held by thread \(i\)
  - 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 \(([\text{Request}_{\text{node}}}] \leq [\text{Avail}])\) {
          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
  - Bridge example: Godzilla picks up a car, hurls it into river. Deadlock solved!
  - But, not always possible: killing thread holding mutex leaves world inconsistent
- Proceed without the resource
  - Requires robust exception handling code
  - E.g., Amazon will say you can buy book, if inventory subsystem doesn’t reply quickly enough (wrong answer quickly is better than right answer slowly)
- Roll back actions of deadlocked threads
  - Hit rewind button, pretend last few minutes never happened
  - Bridge example: make one car roll backwards (may require others behind him)
  - Common technique in databases (transactions)
  - Of course, if you restart in the same way, may reenter deadlock once again
- Many operating systems use other options
Resource Requests Over Time

- Applications usually don’t know exactly when/what they’ll request
- Resources are taken/released over time
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)
  • Often true (most things don't depend on each other) but not very realistic in general

• Don’t allow waiting
  • How phone company avoids deadlock
    • Call someone, either goes through or goes to voicemail
  • Technique used in Ethernet/some multiprocessor nets
    • Everyone speaks at once. On collision, back off and retry
  • Inefficient, since must keep retrying
    • Consider: driving to Toronto, when hit traffic jam, suddenly transported back and told to retry!
Techniques for Preventing Deadlock (cont.)

• 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 bridge at a time

• Force all threads to request resources in fixed 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 by requiring everyone to go clockwise
Banker’s Algorithm

- Invariant: every request always would succeed
- We don’t know order/amount of requests ahead of time
- We assume worst-case max required resource for each thread
- Allow thread i to proceed if

\[(\text{available resources} - \text{request}_i) \geq \text{max remaining required resources by any thread}\]

- Really conservative!
Banker’s Algorithm (cont.)

- Less conservative invariant
  At all times, there exists some order of requests that would succeed

- How to implement this?
  - Allocate resources dynamically
  - Evaluate each request and grant it if some ordering of threads is deadlock free
  - Use deadlock detection algorithm presented earlier
    - BUT: Assume each process needs "max" resources to finish

```plaintext
[Avail] = [FreeResources]
Add all nodes to UNFINISHED
do {
  done = true
  foreach node in UNFINISHED {
    if ([Max_node]-[Alloc_node]<= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_node]
      done = false
    }
  }
} until(done)
```

Each process might need “max” resources in order to finish
Banker’s Algorithm: Key Properties

• Keeps system in “SAFE” state
  • There exists a sequence \( \{T_1, \ldots, T_n\} \) with \( T_1 \) requesting all its remaining resources and finishing, then \( T_2 \) requesting all remaining resources, etc.

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

- Banker’s algorithm with dining politicians
  - “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 politician? 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
    - …
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** (wait indefinitely) versus **deadlock** (circular waiting)

• Four conditions for deadlocks
  • **Mutual exclusion**
    • Only limited number of threads at a time can use resources
  • **Hold and wait**
    • Thread hold at least one resource while waiting to acquire additional ones
  • **No preemption**
    • Resources are released only voluntarily by threads
  • **Circular wait**
    • $\exists$ set \{T\_1, ..., 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** problem and pretend that deadlocks never occur in system
Questions?
Acknowledgment

• Slides by courtesy of Anderson, Culler, Stoica, Silberschatz, Joseph, and Canny