Recall: Real-Time Scheduling (RTS)

- **Hard Real-Time**
  - Attempt to meet all deadlines
  - EDF (Earliest Deadline First), LLF (Least Laxity First), RM (Rate-Monotonic), DM (Deadline Monotonic)
- **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:

```
T1  T2  T3  T4
C_1  C_2  C_3  C_4
D_1  D_2  D_3  D_4
```
Example: Round-Robin Scheduling Doesn’t Work

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

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 tends to 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

Starvation vs 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 $\Rightarrow$ 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 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
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 \( T_1 \).

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 \( T_i \rightarrow R_j \)
  - 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 \( R_j \rightarrow T_i \)

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 \textit{never} enter a deadlock
  - Need to monitor all lock acquisitions
  - Selectively deny those that \textit{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):
  
  \[ [\text{FreeResources}] : \text{Current free resources each type} \]
  
  \[ [\text{Request}_x] : \text{Current requests from thread } x \]
  
  \[ [\text{Alloc}_x] : \text{Current resources held by thread } x \]

- See if tasks can eventually terminate on their own

\[
[\text{Avail}] = [\text{FreeResources}] \\
\text{Add all nodes to UNFINISHED} \}
\]

\[
done = true \}
\]

\[
\text{Foreach node in UNFINISHED } \{
\text{if } ([\text{Request}_\text{node}] \leq [\text{Avail}]) \{
\text{remove node from UNFINISHED} \}
\text{[Avail]} = [\text{Avail}] + [\text{Alloc}_\text{node}] \\
\text{done = false} \}
\}
\]

- 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

Administrivia

- Ion out next week (travelling to China):
  - 11/9: lecture will be given by Anthony Joseph (tentatively)
  - 11/11: lecture will be given by Neeraja

Deadline for 1st midterm regrades: Friday, 10/6

BREAK
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 (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
    - 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 \((\text{[Request]}_n) <= [\text{Avail}]\) {
        - remove node from UNFINISHED
        - \([\text{Avail}] = [\text{Avail}] + [\text{Alloc}]_n\)
      }
    }
  - done = false
  } until(done)

  » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting \((\text{[Max]}_n - [\text{Alloc}]_n)\) for \((\text{[Request]}_n)\) and \([\text{Avail}]\)

  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:
    - \((\text{available resources} - \#\text{requested}) \geq \text{max}\)
      - 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 \((\text{[Max]}_n - [\text{Alloc}]_n)\) for \((\text{[Request]}_n)\) and \([\text{Avail}]\)
  - 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 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 2nd to last, and no one would have \(k-1\)
    - It’s 3rd to last, and no one would have \(k-2\)
    - …
Virtualizing Resources

- Physical Reality:
  Different Processes/Threads share the same hardware
  - Need to multiplex CPU (Just finished: scheduling)
  - Need to multiplex use of Memory (starting today)
  - Need to multiplex disk and devices (later in term)
- Why worry about memory sharing?
  - The complete working state of a process and/or kernel is defined by its data in memory (and registers)
  - Consequently, cannot just let different threads of control use the same memory
    » Physics: two different pieces of data cannot occupy the same locations in memory
  - Probably don’t want different threads to even have access to each other’s memory (protection)

Next Objective

- Dive deeper into the concepts and mechanisms of memory sharing and address translation
- Enabler of many key aspects of operating systems
  - Protection
  - Multi-programming
  - Isolation
  - Memory resource management
  - I/O efficiency
  - Sharing
  - Inter-process communication
  - Debugging
  - Demand paging
- Today: Translation

Recall: Single and Multithreaded Processes

- Threads encapsulate concurrency
  - “Active” component of a process
- Address spaces encapsulate protection
  - Keeps buggy program from trashing the system
  - “Passive” component of a process

Important Aspects of Memory Multiplexing

- Protection:
  - Prevent access to private memory of other processes
    » Different pages of memory can be given special behavior (Read Only, Invisible to user programs, etc).
    » Kernel data protected from User programs
    » Programs protected from themselves
- Controlled overlap:
  - Separate state of threads should not collide in physical memory.
    Obviously, unexpected overlap causes chaos!
  - Conversely, would like the ability to overlap when desired (for communication)
- Translation:
  - Ability to translate accesses from one address space (virtual) to a different one (physical)
  - When translation exists, processor uses virtual addresses, physical memory uses physical addresses
  - Side effects:
    » Can be used to avoid overlap
    » Can be used to give uniform view of memory to programs
Recall: Loading

Binding of Instructions and Data to Memory

Assume 4byte words
0x300 = 4 * 0xC0
0xC0 = 0000 1100 0000
0x300 = 0011 0000 0000

Process view of memory

Second copy of program from previous example

Need address translation!
Second copy of program from previous example

Process view of memory

Physical addresses

<table>
<thead>
<tr>
<th>0x0000</th>
<th>0x0300</th>
<th>0x0900</th>
<th>0x1300</th>
</tr>
</thead>
<tbody>
<tr>
<td>App X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Physical addresses

<table>
<thead>
<tr>
<th>0x1900</th>
<th>8C204C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1904</td>
<td>0C00680</td>
</tr>
<tr>
<td>0x1908</td>
<td>2021FFF</td>
</tr>
<tr>
<td>0x190C</td>
<td>14200642</td>
</tr>
<tr>
<td>0x1910</td>
<td></td>
</tr>
</tbody>
</table>

• One of many possible translations!
• Where does translation take place?
  Compile time, Link/Load time, or Execution time?

Summary

• Real-time scheduling
  – Need to meet a deadline, predictability essential
  – Earliest Deadline First (EDF) and Rate Monotonic (RM) scheduling

• Four requirements for deadlock:
  – Mutual exclusion
  – Hold and wait
  – No preemption
  – Circular wait

• Simple Protection through segmentation
  – Base + Limit registers restrict memory accessible to user
  – Can be used to translate as well