Recall: 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
- Times have deadlines (D) and known computation times (C)
- Example Setup:

```
T1  C1  D1
T2  C2  D2
T3  C3  D3
T4  C4  D4
```

Example: Round-Robin Scheduling Doesn’t Work

```
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

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⇒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 ⇒ 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:

  ```
  Thread A
  x.P();
y.P();
x.V();
y.V();

  Thread B
  y.P();
x.P();
y.V();
x.V();
y.V();
  ```

  - 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 $\Rightarrow$ 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 $T_1$
Symbols

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 $R_j \rightarrow T_i$

Administrivia

- Upcoming deadlines:
  - HW 2 due today 2/29
  - Project 1 final code due Fri 3/4, final report due Mon 3/7

- Midterm next week Wed 3/9 6-7:30 10 Evans and 155 Dwinelle
  - Midterm review session: Sun 3/6 2-5PM at 2060 VLSB
  - Rooms assignment: aa-eh 10 Evans, ej-oa 155 Dwinelle
  - Lectures (including #12), project, homeworks readings, textbook
  - No books, no calculators, one double-side page handwritten notes
  - No class that day, extra office hours

- Use Piazza private posts only for administrative questions

- Apple Core OS Tech Talk Infosession tomorrow 6:15P in Woz

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

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

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

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 (con’t)

- 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_node] - [Alloc_node] ≤ [Avail]) for ([Request_node] ≤ [Avail])
    Grant request if result is deadlock free (conservative!)
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
    \((\text{Max}_\text{node} - \text{Alloc}_\text{node}) \leq [\text{Avail}])\) for \((\text{Request}_\text{node} \leq [\text{Avail}])\)
  » Grant request if result is deadlock free (conservative!)
  » 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 for Preventing Deadlock

Virtualizing Resources

- Physical Reality:
  Different Processes/Threads share the same hardware
  - Need to multiplex CPU (just finished: scheduling)
  - Need to multiplex use of Memory (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: Linking, Segmentation

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

- 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
- 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
Binding of Instructions and Data to Memory

Process view of memory

<table>
<thead>
<tr>
<th>Data1: dw 32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start: lw r1,0(data1)</td>
</tr>
<tr>
<td>Loop: addi r1, r1, -1</td>
</tr>
<tr>
<td>Bnz r1, loop</td>
</tr>
<tr>
<td>Checkit: ...</td>
</tr>
</tbody>
</table>

Physical memory

<table>
<thead>
<tr>
<th>Memory Address: 0x0300, 0x0900</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value: 8C2000C0, 00000280</td>
</tr>
</tbody>
</table>

Physical addresses

<table>
<thead>
<tr>
<th>Memory Address: 0x01300, 0x01900</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value: 8C2004C0, 00000680</td>
</tr>
</tbody>
</table>

Second copy of program from previous example

Process view of memory

<table>
<thead>
<tr>
<th>Data1: dw 32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start: lw r1,0(data1)</td>
</tr>
<tr>
<td>Loop: addi r1, r1, -1</td>
</tr>
<tr>
<td>Bnz r1, r0, loop</td>
</tr>
<tr>
<td>Checkit: ...</td>
</tr>
</tbody>
</table>

Physical addresses

<table>
<thead>
<tr>
<th>Memory Address: 0x00000020, 0x000000FF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value: 8C2000C0, 00000280</td>
</tr>
</tbody>
</table>

Compile time, Link/Load time, or Execution time?

- One of many possible translations!
- Where does translation take place?
Multi-step Processing of a Program for Execution

• Preparation of a program for execution involves components at:
  – Compile time (i.e., “gcc”)
  – Link/Load time (UNIX “ld” does link)
  – Execution time (e.g., dynamic libs)

• Addresses can be bound to final values anywhere in this path
  – Depends on hardware support
  – Also depends on operating system

• Dynamic Libraries
  – Linking postponed until execution
  – Small piece of code, stub, used to locate appropriate memory-resident library routine
  – Stub replaces itself with the address of the routine, and executes routine

Recall: Uniprogramming

• Uniprogramming (no Translation or Protection)
  – Application always runs at same place in physical memory since only one application at a time
  – Application can access any physical address
    - Application given illusion of dedicated machine by giving it reality of a dedicated machine

Multiprogramming (primitive stage)

• Multiprogramming without Translation or Protection
  – Must somehow prevent address overlap between threads
    - Everything adjusted to memory location of program (loads, stores, jumps)
    - Translation done by a linker-loader (relocation)
    - Common in early days (… till Windows 3.x, 95?)
  – With this solution, no protection: bugs in any program can cause other programs to crash or even the OS
Multiprogramming (Version with Protection)

- Can we protect programs from each other without translation?
  - Yes: use two special registers `BaseAddr` and `LimitAddr` to prevent user from straying outside designated area
  - During switch, kernel loads new base/limit from PCB (Process Control Block)
    » User not allowed to change base/limit registers

Recall: General Address translation

- Recall: Address Space:
  - All the addresses and state a process can touch
  - Each process and kernel has different address space
- Consequently, two views of memory:
  - View from the CPU (what program sees, virtual memory)
  - View from memory (physical memory)
  - Translation box (MMU) converts between the two views
- Translation makes it much easier to implement protection
  - If task A cannot even gain access to task B’s data, no way for A to adversely affect B
- With translation, every program can be linked/loaded into same region of user address space

Simple Example: Base and Bounds (CRAY-1)

- Could use base/bounds for dynamic address translation – translation happens at execution:
  - Alter address of every load/store by adding “base”
  - Generate error if address bigger than limit
- This gives program the illusion that it is running on its own dedicated machine, with memory starting at 0
  - Program gets continuous region of memory
  - Addresses within program do not have to be relocated when program placed in different region of DRAM

Issues with Simple B&B Method

- Fragmentation problem over time
  - Not every process is same size ➞ memory becomes fragmented
- Missing support for sparse address space
  - Would like to have multiple chunks/program (Code, Data, Stack)
- Hard to do inter-process sharing
  - Want to share code segments when possible
  - Want to share memory between processes
  - Helped by providing multiple segments per process
Running more programs than fit in memory: Swapping

- Q: What if not all processes fit in memory?
- A: Swapping: Extreme form of Context Switch
  - In order to make room for next process, some or all of the previous process is moved to disk
  - This greatly increases the cost of context-switching

Desirable alternative?
- Some way to keep only active portions of a process in memory at any one time
- Need finer granularity control over physical memory

---

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
    » ∃ 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 \textit{never} enter a deadlock
  - Ignore the problem and pretend that deadlocks never occur in system

---

Summary (2)

- Memory is a resource that must be multiplexed
  - Controlled Overlap: only shared when appropriate
  - Translation: Change virtual addresses into physical addresses
  - Protection: Prevent unauthorized sharing of resources
- Simple Protection through segmentation
  - Base + Limit registers restrict memory accessible to user
  - Can be used to translate as well