Last Class: Deadlocks

• Necessary conditions for deadlock:
  – Mutual exclusion
  – Hold and wait
  – No preemption
  – Circular wait

• Ways of handling deadlock
  – Deadlock detection and recovery
  – Deadlock prevention
  – Deadlock avoidance

Deadlock Prevention with Resource Reservation

• Threads provide advance information about the maximum resources they may need during execution
• Define a sequence of threads \( \{t_1, ..., t_n\} \) as safe if for each \( t_i \), the resources that \( t_i \) can still request can be satisfied by the currently available resources plus the resources held by all \( t_j, j < i \).
• A safe state is a state in which there is a safe sequence for the threads.
• An unsafe state is not equivalent to deadlock, it just may lead to deadlock, since some threads might not actually use the maximum resources they have declared.
• Grant a resource to a thread is the new state is safe
• If the new state is unsafe, the thread must wait even if the resource is currently available.
• This algorithm ensures no circular-wait condition exists.
Example

• Threads \( t_1 \), \( t_2 \), and \( t_3 \) are competing for 12 tape drives.
• Currently, 11 drives are allocated to the threads, leaving 1 available.
• The current state is safe (there exists a safe sequence, \( \{ t_1, t_2, t_3 \} \) where all threads may obtain their maximum number of resources without waiting)
  - \( t_1 \) can complete with the current resource allocation
  - \( t_2 \) can complete with its current resources, plus all of \( t_1 \)'s resources, and the unallocated tape drive.
  - \( t_3 \) can complete with all its current resources, all of \( t_1 \) and \( t_2 \)'s resources, and the unallocated tape drive.

<table>
<thead>
<tr>
<th></th>
<th>max need</th>
<th>in use</th>
<th>could want</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 )</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>( t_2 )</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>( t_3 )</td>
<td>12</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

Example (contd)

• If \( t_3 \) requests one more drive, then it must wait because allocating the drive would lead to an unsafe state.
• There are now 0 available drives, but each thread might need at least one more drive.

<table>
<thead>
<tr>
<th></th>
<th>max need</th>
<th>in use</th>
<th>could want</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 )</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>( t_2 )</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>( t_3 )</td>
<td>12</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>
Deadlock Avoidance using Resource Allocation Graph

- Claim edges: an edge from a thread to a resource that may be requested in the future
- Satisfying a request results in converting a claim edge to an allocation edge and changing its direction.
- A cycle in this extended resource allocation graph indicates an unsafe state.
- If the allocation would result in an unsafe state, the allocation is denied even if the resource is available.
  - The claim edge is converted to a request edge and the thread waits.
- This solution does not work for multiple instances of the same resource.

![Resource Allocation Graph]

Banker's Algorithm

- This algorithm handles multiple instances of the same resource.
- Force threads to provide advance information about what resources they may need for the duration of the execution.
- The resources requested may not exceed the total available in the system.
- The algorithm allocates resources to a requesting thread if the allocation leaves the system in a safe state.
- Otherwise, the thread must wait.
Preventing Deadlock with Banker's Algorithm

class ResourceManager {
    int n;        // # threads
    int m;      // # resources
    int avail[m], // # of available resources of each type
    max[n,m],   // # of each resource that each thread may want
    alloc[n,m], // # of each resource that each thread is using
    need[n,m],   // # of resources that each thread might still request

    public void synchronized allocate (int request[m], int i) {
        // request contains the resources being requested
        // i is the thread making the request

        if (request > need[i]) //vector comparison
            error(); // Can't request more than you declared
        else while (request[i] > avail)
            wait(); // Insufficient resources available

        // enough resources exist to satisfy the requests
        // See if the request would lead to an unsafe state
        avail = avail - request; // vector additions
        alloc[i] = alloc[i] + request;
        need[i] = need[i] - request;

        while ( !safeState () ) { // if this is an unsafe state, undo the allocation and wait
            <undo the changes to avail, alloc[i], and need[i]>
            wait ();
            <redo the changes to avail, alloc[i], and need[i]>
        }
}
Banker's Algorithm: Safety Check

private boolean safeState () {
    boolean work[m] = avail[m]; // accommodate all resources
    boolean finish[n] = false; // none finished yet

    // find a process that can complete its work now
    while (find i such that finish[i] == false
        and need[i] <= work) { // vector operations
        work = work + alloc[i]
        finish[i] = true;
    }

    if (finish[i] == true for all i)
        return true;
    else
        return false;
}

• Worst case: requires $O(mn^2)$ operations to determine if the system is safe.

Example using Banker's Algorithm

System snapshot:

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>Allocation</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>P₀</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P₁</td>
<td>1</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>P₂</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>P₃</td>
<td>0</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Total</td>
<td>2</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>
Example (contd)

• How many resources are there of type (A,B,C)?

• What is the contents of the Need matrix?

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₀</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P₁</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P₃</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Is the system in a safe state? Why?

Example: solutions

• How many resources of type (A,B,C)?  (3,14,11)
resources = alloc + avail
• What is the contents of the need matrix?
Need = Max - Allocation.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₀</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P₁</td>
<td>0</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>P₂</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P₃</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

• Is the system in a safe state? Why?
• Yes, because the processes can be executed in the sequence P₀, P₂, P₁, P₃, even if each process asks for its maximum number of resources when it executes.
Example (contd)

- If a request from process P₁ arrives for additional resources of (0,5,2), can the Banker's algorithm grant the request immediately?
- What would be the new system state after the allocation?

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P₀</td>
<td>0 0 1</td>
<td></td>
</tr>
<tr>
<td>P₁</td>
<td>1 7 5</td>
<td></td>
</tr>
<tr>
<td>P₂</td>
<td>2 3 5</td>
<td></td>
</tr>
<tr>
<td>P₃</td>
<td>0 6 5</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- What is a sequence of process execution that satisfies the safety constraint?

Example: solutions

- If a request from process P₁ arrives for additional resources of (0,5,2), can the Banker's algorithm grant the request immediately? Show the system state, and other criteria.
  Yes. Since
  1. (0,5,2) ≤ (1,5,2), the Available resources, and
  2. (0,5,2) + (1,0,0) = (1,5,2) ≤ (1,7,5), the maximum number P₁ can request.
  3. The new system state after the allocation is:

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Max</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P₀</td>
<td>0 0 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>P₁</td>
<td>1 5 2</td>
<td>1 7 5</td>
</tr>
<tr>
<td>P₂</td>
<td>1 3 5</td>
<td>2 3 5</td>
</tr>
<tr>
<td>P₃</td>
<td>0 6 3</td>
<td>0 6 5</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>1 0 0</td>
</tr>
</tbody>
</table>

and the sequence P₀, P₂, P₁, P₃ satisfies the safety constraint.
Summary

• Deadlock: situation in which a set of threads/processes cannot proceed because each requires resources held by another member of the set.
• Detection and recovery: recognize deadlock after it has occurred and break it.
• Avoidance: don't allocate a resource if it would introduce a cycle.
• Prevention: design resource allocation strategies that guarantee that one of the necessary conditions never holds
• Code concurrent programs very carefully. This only helps prevent deadlock over resources managed by the program, not OS resources.
• Ignore the possibility! (Most OSes use this option!!)

Computing Parable

• The Donkey that starved
Where we are in the course

• Discussed:
  – Processes & Threads
  – CPU Scheduling
  – Synchronization & Deadlock

• Next up:
  – Memory Management

• Yet to come:
  – File Systems and I/O Storage
  – Distributed Systems

Memory Management

• Where is the executing process?

• How do we allow multiple processes to use main memory simultaneously?

• What is an address and how is one interpreted?
Background: Computer Architecture

- Program executable starts out on disk
- The OS loads the program into memory
- CPU fetches instructions and data from memory while executing the program

Memory Management: Terminology

- **Segment**: A chunk of memory assigned to a process.
- **Physical Address**: A real address in memory
- **Virtual Address**: An address relative to the start of a process's address space.
Where do addresses come from?

How do programs generate instruction and data addresses?

• **Compile time:** The compiler generates the exact physical location in memory starting from some fixed starting position $k$. The OS does nothing.

• **Load time:** Compiler generates an address, but at load time the OS determines the process' starting position. Once the process loads, it does not move in memory.

• **Execution time:** Compiler generates an address, and OS can place it anywhere it wants in memory.

Uniprogramming

• OS gets a fixed part of memory (highest memory in DOS).
• One process executes at a time.
• Process is always loaded starting at address 0.
• Process executes in a contiguous section of memory.
• Compiler can generate physical addresses.
• Maximum address = Memory Size - OS Size
• OS is protected from process by checking addresses used by process.
Uniprogramming

⇒ Simple, but does not allow for overlap of I/O and computation.

Multiple Programs Share Memory

Transparency:
- We want multiple processes to coexist in memory.
- No process should be aware that memory is shared.
- Processes should not care what physical portion of memory they are assigned to.

Safety:
- Processes must not be able to corrupt each other.
- Processes must not be able to corrupt the OS.

Efficiency:
- Performance of CPU and memory should not be degraded badly due to sharing.
Relocation

- Put the OS in the highest memory.
- Assume at compile/link time that the process starts at 0 with a maximum address = memory size - OS size.
- Load a process by allocating a contiguous segment of memory in which the process fits.
- The first (smallest) physical address of the process is the base address and the largest physical address the process can access is the limit address.

Relocation

- **Static Relocation:**
  - at load time, the OS adjusts the addresses in a process to reflect its position in memory.
  - Once a process is assigned a place in memory and starts executing it, the OS cannot move it. (Why?)

- **Dynamic Relocation:**
  - hardware adds relocation register (base) to virtual address to get a physical address;
  - hardware compares address with limit register (address must be less than limit).
  - If test fails, the processor takes an address trap and ignores the physical address.
Dynamic Relocation

• **Advantages:**
  – OS can easily move a process during execution.
  – OS can allow a process to grow over time.
  – Simple, fast hardware: two special registers, an add, and a compare.

• **Disadvantages:**
  – Slows down hardware due to the add on every memory reference.
  – Can't share memory (such as program text) between processes.
  – Process is still limited to physical memory size.
  – Degree of multiprogramming is very limited since all memory of all active processes must fit in memory.
  – Complicates *memory management*.

Relocation: Properties

• **Transparency:** processes are largely unaware of sharing.

• **Safety:** each memory reference is checked.

• **Efficiency:** memory checks and virtual to physical address translation are fast as they are done in hardware, BUT if a process grows, it may have to be moved which is very slow.
Memory Management: Memory Allocation

As processes enter the system, grow, and terminate, the OS must keep track of which memory is available and utilized.

- **Holes:** pieces of free memory (shaded above in figure)
- Given a new process, the OS must decide which hole to use for the process

### Memory Allocation Policies

- **First-Fit:** allocate the first one in the list in which the process fits. The search can start with the first hole, or where the previous first-fit search ended.
- **Best-Fit:** Allocate the smallest hole that is big enough to hold the process. The OS must search the entire list or store the list sorted by size hole list.
- **Worst-Fit:** Allocate the largest hole to the process. Again the OS must search the entire list or keep the list sorted.
- Simulations show first-fit and best-fit usually yield better storage utilization than worst-fit; first-fit is generally faster than best-fit.
Fragmentation

- **External Fragmentation**
  - Frequent loading and unloading programs causes free space to be broken into little pieces
  - External fragmentation exists when there is enough memory to fit a process in memory, but the space is not contiguous
  - *50-percent rule*: Simulations show that for every $2N$ allocated blocks, $N$ blocks are lost due to fragmentation (i.e., $1/3$ of memory space is wasted)
  - We want an allocation policy that minimizes wasted space.

- **Internal Fragmentation**:
  - Consider a process of size 8846 bytes and a block of size 8848 bytes
  - It is more efficient to allocate the process the entire 8848 block than it is to keep track of 2 free bytes
  - Internal fragmentation exists when memory internal to a partition that is wasted