Addressing Deadlock

- **Prevention**
  - Design the system so that deadlock is impossible

- **Avoidance**
  - Construct a model of system states, then choose a strategy that, when resources are assigned to processes, will not allow the system to go to a deadlock state

- **Detection & Recovery**
  - Check for deadlock (periodically or sporadically) and identify and which processes and resources involved
  - Recover by killing one of the deadlocked processes and releasing its resources

- **Manual intervention**
  - Have the operator reboot the machine if it seems too slow
Deadlock Avoidance

- Deadlock prevention
  - Assumes all resources are requested at start time

- Realistic scenarios
  - Resources are requested incrementally

- Deadlock Avoidance: Basic idea
  - Try to see the worst that could happen
  - Do not grant an incremental resource request to a process if this allocation might lead to deadlock
  - Conservative/pessimistic approach
Deadlock Avoidance

- Assume OS knows
  - Number of available instances of each resource
    - Mutex: a resource with one instance available
    - Semaphore: a resource with possibly multiple “instances” available
  - For each process
    - Current amount of each resource it owns
    - Maximum amount of each resource it might ever need
      - For a mutex this means: Will the process ever lock the mutex?

- Assume processes are independent
  - If one blocks, others can finish if they have enough resources
Deadlock and Resources

- Single instance of each resource
  - Find cycle in resource allocation graph

- Multiple instance of each resource
  - Process can request any number of instances for a given resource
    - May only use some of them
Deadlock Avoidance: Safe vs. Unsafe

**Approach**
- Define a model of system states (SAFE, UNSAFE)
- Choose a strategy that guarantees that the system will not go to a deadlock state

**Safe**
- **Guarantee**
  - There is some scheduling order in which every process can run to completion even if all of them suddenly and simultaneously request their maximum number of resources
  - From a safe state
    - The system can guarantee that all processes will finish
Deadlock Avoidance: Safe vs. Unsafe

**Approach**
- Define a model of system states (SAFE, UNSAFE)
- Choose a strategy that guarantees that the system will not go to a deadlock state

**Unsafe state: no such guarantee**
- A deadlock state is an unsafe state
- An unsafe state may not be a deadlock state
- Some process may be able to complete
Safe vs. Unsafe

- Safe
  - There is a way for all processes to finish executing without deadlock.

- Goal
  - Guide the system down one of those paths successfully.

All states

Safe

Unsafe

Deadlocked
How to guide the system down a safe path of execution

- New function: is a given state **safe**?
- When a resource allocation request arrives
  - Pretend that we approve the request
    - Call function: Would we then be safe?
  - If safe
    - Approve request
  - Otherwise
    - Block process until its request can be safely approved
Is a state safe?

- What is a “state”?
  - For each resource,
    - Current amount **available**
    - Current amount **allocated** to each process
    - Future amount **needed** by each process

Semaphore s  Mutex m

<table>
<thead>
<tr>
<th>Free</th>
<th>P1 alloc</th>
<th>P2 alloc</th>
<th>P1 need</th>
<th>P2 need</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Copyright ©: University of Illinois CS 241 Staff
Is a state safe?

- Safe
  - There is an execution order that can finish

- Pessimistic assumption
  - Processes never release resources until they’re done
Is a state safe?

- Safe
  - There is an execution order that can finish
    - P1 can finish using what it has plus what’s free
    - P2 can finish using what it has plus what’s free, plus what P1 will release when it finishes
    - P3 can finish using what it has, plus what P1 and P2 will release when they finish
    - ...

Copyright ©: University of Illinois CS 241 Staff
Is a state safe?

Search for an order \( P_1, P_2, P_3, \ldots \) such that:

- \( P_1 \)'s max resource needs \( \leq \) what it has + what's free
- \( P_2 \)'s max resource needs \( \leq \) what it has + what's free + what \( P_1 \) will release when it finishes
- \( P_3 \)'s max resource needs \( \leq \) what it has + what's free + what \( P_1 \) and \( P_2 \) will release when they finish

How do we figure that out?
Try all orderings?
How many orderings do we need to find?
Inspiration...
Playing pickup sticks with processes

- **Pick up**
  - Find a stick on top
    - Find a process that can finish with what it has plus what’s free
  - Remove stick
    - Process releases its resources

- **Repeat**
  - Until all processes have finished
    - Answer: **safe**
  - Or we get stuck
    - Answer: **unsafe**
Try it: is this state safe?

Semaphore s
- Free
- P1 alloc
- P2 alloc
- P1 need
- P2 need

Mutex M

Which process can go first?
Example 2: Is this state safe?

Semaphore s

Can P1 go first?

Can P2 go first?

Can P3 go first?

Free

P1 alloc

P2 alloc

P3 alloc

P1 need

P2 need

P3 need
How to guide the system down a safe path of execution

- New function: is a given state safe?
- When a resource allocation request arrives
  - Pretend that we approve the request
    - Call function: Would we then be safe?
  - If safe
    - Approve request
  - Otherwise
    - Block process until its request can be safely approved

Banker’s Algorithm
Banker’s Algorithm

- Dijkstra, 1965
  - Each customer tells banker the maximum number of resources it needs, before it starts
  - Customer borrows resources from banker
  - Customer returns resources to banker
  - Banker only lends resources if the system will stay in a safe state after the loan
- Customer may have to wait
Banker’s Algorithm: Take 1

For each request
- If approved, would we still be safe?
  - If yes
    - Approve
  - If no
    - Block

<table>
<thead>
<tr>
<th>Semaphore s</th>
<th>Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free</td>
<td></td>
</tr>
<tr>
<td>P1 alloc</td>
<td></td>
</tr>
<tr>
<td>P2 alloc</td>
<td></td>
</tr>
<tr>
<td>P1 need</td>
<td></td>
</tr>
<tr>
<td>P2 need</td>
<td></td>
</tr>
</tbody>
</table>
Banker’s Algorithm: Take 2

```c
mutex m1, m2;
int x, y;

while (1) {
    lock(m1);
    x++;
    unlock(m1);

    lock(m2);
    y++;
    unlock(m2);
}
```

<table>
<thead>
<tr>
<th>m1</th>
<th>m2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free</td>
<td>Free</td>
</tr>
</tbody>
</table>

- P1 alloc
- P2 alloc
- P1 need
- P2 need
Banker’s algorithm example 2

All states

Safe

Unsafe

Deadlocked
Formalized Banker’s Algorithm

Given
- n resource types
- P processes
- p.Max[1...n]
  - Maximum number of resource i needed by p
- p.Alloc[i]
  - Number of instances of resource i held by p
  - <= p.Max[i]
- Avail[1...n]
  - Current number of available resources of each type

Algorithm:

while (there exists a p in P such that {for all i (p.Need[i] <= Available[i])}) {
    for (all i) {
        Avail[i] += p.Alloc[i];
        P = P - p;
    }
}

If P is empty then system is safe
Current Allocation

<table>
<thead>
<tr>
<th>Pr</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>P0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>3</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>Available</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>A</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

Can P1 request (A:1 B:0 C:2)?
New Allocation

<table>
<thead>
<tr>
<th>Pr</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>P0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>P1</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Can P0 request (A:0 B:2 C:0)?
Outcome

- P0’s request for 2 Bs
  - Cannot be granted because
    - Would prevent any other process from completing if they need their maximum claim
- Just Because It’s Unsafe Doesn’t mean it will always deadlock
  - P0 could have been allocated 2 Bs and a deadlock might not have occurred if:
    - P2 didn’t use its maximum resources but finished using the resources it had
Concluding notes

- In general, deadlock detection or avoidance is expensive.
- Must evaluate cost and frequency of deadlock against costs of detection or avoidance.
- Deadlock avoidance and recovery may cause indefinite postponement.
- Unix, Windows use Ostrich Algorithm (do nothing).
- Typical apps use deadlock prevention (order locks).
- Transaction systems (e.g., credit card systems) need to use deadlock detection/recovery/avoidance/prevention (why?)