Deadlocks
Addressing Deadlock

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

- **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

- **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

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

- **Deadlock detection**
  - 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

- 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
  - To solve deadlock avoidance, we need
    - Current number of available instances of each resource
    - Maximum number of each resource needed for each process
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

- 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 deadlocking

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

<table>
<thead>
<tr>
<th>Memory</th>
<th>GPU</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>

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?

- Safe
  - There is an execution order that can 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?

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

Can P1 go first?

Can P2 go first?

Can P3 go first?
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>Memory</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>
mutex m1, m2;
int x, y;

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

    lock(m2);
    y++;
    unlock(m2);
}
Banker’s algorithm example 2
Given
- \( n \) resource types
- \( P \) processes
- \( p.\text{Max}[1...n] \)
  - Maximum number of resource \( i \) needed by \( p \)
- \( p.\text{Alloc}[i] \)
  - Number of instances of resource \( i \) held by \( p \)
  - \( \leq p.\text{Max}[i] \)
- \( \text{Avail}[1...n] \)
  - Current number of available resources of each type
- \( p.\text{Need}[i] = p.\text{MAX}[i] - p.\text{Alloc}[i] \)

Algorithm:

```
while (there exists a \( p \) in \( P \) such that \{for all \( i \) \( (p.\text{Need}[i] \leq \text{Available}[i]) \}) \} *

for (all \( i \)) {
    \text{Avail}[i] += p.\text{Alloc}[i];
    \text{P} = \text{P} - p;
}
```

If \( P \) is empty then system is safe
Can P1 request (A:1 B:0 C:2) ?

<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></td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

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