Deadlock Modelling

- Processes and resources can be modelled with directed graphs:

  - Resource R is held by Process A
  - Process B requests Resource S
  - Deadlock: processes request resources they can never get
Deadlock Modelling

A:
- Request R
- Request S
- Release R
- Release S

B:
- Request S
- Request T
- Release S
- Release T

C:
- Request T
- Request R
- Release T
- Release R

A requests R
B requests S
C requests T
A requests S
B requests T
C requests R

Deadlock!
Strategies for Deadlock

Strategies for dealing with Deadlocks
1. Ignore the problem altogether
2. Detection and recovery
3. Dynamic avoidance
   - Careful resource allocation
4. Prevention
   - Negating one of the four necessary conditions

Solution 1: The Ostrich Algorithm

- Pretend there is no problem.
- This is reasonable if deadlocks occur very rarely, and the cost of prevention is high.
- UNIX and Windows take this approach.
- It is a trade off between
  - Convenience
  - Correctness
Solution 2: Detection and Recovery

- Detect that a deadlock has occurred.
- Initiate a recovery algorithm to break the deadlock.

Detection with One Resource of Each Type

- Note the resource ownership and requests.
- Perform a depth-first search within the resource graph, searching for any cycles.
- If any cycle exists in the graph, then parts of the system are deadlocked.
Detection with Multiple Resources of Each Type

Resources in existence
\((E_1, E_2, E_3, \ldots, E_n)\)

Current allocation matrix
\[
\begin{bmatrix}
C_{11} & C_{12} & \cdots & C_{1n} \\
C_{21} & C_{22} & \cdots & C_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
C_{n1} & C_{n2} & \cdots & C_{nn}
\end{bmatrix}
\]
Row \(n\) is current allocation to process \(n\)

Resources available
\((A_1, A_2, A_3, \ldots, A_m)\)

Request matrix
\[
\begin{bmatrix}
R_{11} & R_{12} & \cdots & R_{1m} \\
R_{21} & R_{22} & \cdots & R_{2m} \\
\vdots & \vdots & \ddots & \vdots \\
R_{n1} & R_{n2} & \cdots & R_{nm}
\end{bmatrix}
\]
Row 2 is what process 2 needs

Data structures needed by deadlock detection algorithm

Detection with Multiple Resources of Each Type (2)

Existing Resources

<table>
<thead>
<tr>
<th>memory</th>
<th>tape drives</th>
<th>swap</th>
<th>printer</th>
</tr>
</thead>
<tbody>
<tr>
<td>E = (4  2  3  1)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Available Resources

<table>
<thead>
<tr>
<th>memory</th>
<th>tape drives</th>
<th>swap</th>
<th>printer</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = (2  1  0  0)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Current Allocation Matrix
\[
\begin{bmatrix}
0 & 0 & 1 & 0 \\
2 & 0 & 0 & 1 \\
0 & 1 & 2 & 0
\end{bmatrix}
\]

Request Matrix
\[
\begin{bmatrix}
2 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 \\
2 & 1 & 0 & 0
\end{bmatrix}
\]
Recovery from Deadlock

- **Recovery through preemption**
  - Take a resource from some other process
  - Depends on nature of the resource

- **Recovery through rollback**
  - Checkpoint a process periodically
  - Use this saved state
  - Restart the process from the saved state if it is found deadlocked.

- **Recovery through killing processes**
  - Crudest but simplest way to break a deadlock
  - Kill one of the processes in the deadlock cycle
  - The other processes get its resources
  - Choose process that can be rerun from the beginning

---

Solution 3: Deadlock Avoidance

**Resource Trajectories**

Process 2 requests the printer. Should OS grant the request?
Demonstration of a Safe State

**Banker's Algorithm:**
**Avoiding Unsafe States**

A makes a request. Should the OS grant the request?

No further process can complete, so granting the request is unsafe.
Banker's Algorithm for Multiple Resources

- In each step, find a row whose requests can all be simultaneously satisfied from the available resources.
- If no such row exists, the starting state was unsafe.

<table>
<thead>
<tr>
<th>Process</th>
<th>Tape drives</th>
<th>Printers</th>
<th>Scanners</th>
<th>CD ROMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process</th>
<th>Tape drives</th>
<th>Printers</th>
<th>Scanners</th>
<th>CD ROMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Solution 4: Deadlock Prevention

- **Serially reusable resources:**
  The processes share resources that they use under mutual exclusion

- **Incremental acquisition:**
  Processes hold on to resources already allocated to them while waiting to acquire additional resources.

- **No pre-emption:**
  Once acquired, resources cannot be forcibly withdrawn

- **Wait-for cycle:**
  A circular chain of processes exists, with each holding a resource that the next process in the cycle is waiting to acquire.
Deadlock Prevention:
Attacking the Mutual Exclusion Condition

- Some devices (such as printer) can be spooled
  - Only the printer daemon uses printer resource.
  - Eliminates deadlock for printer.
- Principle:
  - Avoid assigning a resource when it’s not absolutely necessary.
  - As few processes as possible directly claim the resource.
- Not all devices can be spooled

Deadlock Prevention:
Attacking the Hold and Wait Condition

- Require processes to request all it’s resources before starting.
  - A process never has to wait for what it needs
- Problems
  - May not know required resources at start of run.
  - Also ties up resources other processes could be using.
- Variation:
  - To gain a new resource, a process must first give up all resources it holds.
  - Then re-request all resources it immediately needs
Deadlock Prevention:
Attacking the No Pre-emption Condition

- This is rarely a viable option
- Consider a process given the printer
  - Halfway through its job
  - Now forcibly take away printer
  - !?!!

Deadlock Prevention:
Attacking the Circular Wait Condition

- Assign a single numeric ordering to the resources.
- A process that wishes to claim multiple resources must claim them in numeric order.

![Diagram of A and B with arrows i to j]

- If i < j, then B cannot claim i, so no cycle can exist.
- If i > j, then A cannot claim j, so no cycle can exist.

Problem: no single unique ordering may exist that suits everyone.
Deadlock Prevention:

Summary

<table>
<thead>
<tr>
<th>Condition</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutual exclusion</td>
<td>Spool everything</td>
</tr>
<tr>
<td>Hold and wait</td>
<td>Request all resources initially</td>
</tr>
<tr>
<td>No preemption</td>
<td>Take resources away</td>
</tr>
<tr>
<td>Circular wait</td>
<td>Order resources numerically</td>
</tr>
</tbody>
</table>

Summary of approaches to deadlock prevention

Databases:

Two-Phase Locking

- Phase One
  - Process tries to lock all records it needs, one at a time
  - If a needed record is found locked, unlock all and start again.
  - (no real work done in phase one)
- If phase one succeeds, it starts second phase,
  - Performing database updates
  - Releasing locks
- Not a general solution:
  - Only works where process can be stopped and restarted during the first phase without undesirable side effects.