Deadlock Detection and Recovery
  o System does not try to prevent deadlocks
    ▪ Lets deadlocks occur, tries to detect them, then takes some action
    ▪ Use resource allocation graph and detect cycles to determine if there is currently a deadlock
  o Detection with one resource of each type
    ▪ A system can only have one scanner, one CD device, one plotter, etc.
      ▪ Having multiple resources of each type makes another approach necessary
    ▪ More complex example
      ▪ Seven processes (A through G) and six resources (R through W)
        o Process A holds R and wants S
        o Process B holds nothing but wants T
        o Process C holds nothing but wants S
        o Process D holds U and wants S and T
        o Process E holds T and wants V
        o Process F holds W and wants S
        o Process G holds V and wants U
  o Is the system deadlocked, and which processes are involved?
    ▪ Construct resource allocation graph, check for cycles

Visual inspection shows that D, E, and G are deadlocked
Further, A, C, and F are not
  ▪ S can be allocated to any of them, which will then use it until finishing
  ▪ It can then be given to the others in turn without causing deadlock
We need a formal algorithm, as visual inspection is not sufficient
- Many algorithms for detecting cycles in a directed graph are known
  - Below is a simple (inefficient) example
- Perform DFS starting from each node in the graph
  - Maintain a list $L$ of nodes in the current traversal
  - Mark arcs as traversed so we do not repeat
  - If a node appears in $L$ twice, there is a cycle
- Algorithm
  - For each node $N$ in the graph, do the following
    - Initialize $L$ to empty and all arcs as unmarked
    - Add current node to $L$ and check to see if it appears in $L$ twice
      - If it does, there is a cycle
    - If there are unmarked outgoing arcs
      - Pick random unmarked arc and follow it to the new current node, then repeat last step.
    - Else
      - If initial node, no cycles are found and algorithm terminates
      - Else remove the current node and go back to the previous one (backtrack)
- Algorithm run on previous example
  - Start with $R$: $L = [R,A,S]$ before backtracking and not discovering a cycle
  - Start with $A$: $L = [A,S]$ before backtracking and not discovering a cycle
  - Start with $B$:
    - $L = [B,T,E,V,G,U,D]$ if $S$ is chosen, it dead ends and backtracks to $D$
    - Finally, $T$ is added again to get $L = [B,T,E,V,G,U,D,T]$, which has a cycle and terminates the algorithm
- Detection with Multiple Resources of Each Type
  - Introduction
    - Sometimes multiple copies of a resource exist and it’s not necessary to distinguish between the two (e.g., two network devices)
    - We need a different algorithm to solve this problem
  - Matrix-based algorithm
    - $n$ processes labeled $P_1$ through $P_n$
    - $E$ is a vector of existing resources, with $E_i$ resources of each class
Available resource vector – resources that are not currently assigned

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

Current allocation matrix – \(i\)-th row of \(C\) tells how many instances of each resource class \(P_i\) currently holds

\[
\begin{bmatrix}
C_{11} & C_{12} & \cdots & C_{1m} \\
C_{21} & C_{22} & \cdots & C_{2m} \\
\vdots & \vdots & \ddots & \vdots \\
C_{n1} & C_{n2} & \cdots & C_{nm}
\end{bmatrix}
\]

Row \(n\) is current allocation to process \(n\)

Request matrix – \(i\)-th row of \(R\) tells how many instances of each resource class \(P_i\) wants.

\[
\begin{bmatrix}
R_{11} & R_{12} & \cdots & R_{1m} \\
R_{21} & R_{22} & \cdots & R_{2m} \\
\vdots & \vdots & \ddots & \vdots \\
R_{n1} & R_{n2} & \cdots & R_{nm}
\end{bmatrix}
\]

Row \(2\) is what process \(2\) needs

Invariant – Sum of allocations and available resources must equal resources in existence

Algorithm
- Each process initially starts as unmarked
- Look for unmarked process \(P_i\) for which every element in the \(i\)-th row of \(R\) is less than the corresponding element in \(A\)
Implication – this process’s resource needs can be filled, so it should be allocated the resources and allowed to finish.

- If one is found, add the \( i \)-th row of \( C \) to \( A \)
  - After the process finishes, all of its resources will be made available again.
- If no found, terminate the algorithm
  - All unmarked processes are deadlocked

Example – 3 processes and 4 resource classes

<table>
<thead>
<tr>
<th>Tape drives</th>
<th>Plotters</th>
<th>Scanners</th>
<th>CD Roms</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

\( E = \begin{bmatrix} 4 & 2 & 3 & 1 \end{bmatrix} \)

<table>
<thead>
<tr>
<th>Tape drives</th>
<th>Plotters</th>
<th>Scanners</th>
<th>CD Roms</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

\( A = \begin{bmatrix} 2 & 1 & 0 & 0 \end{bmatrix} \)

\( C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix} \)

\( R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix} \)

- Initially, the third process can be run, yielding \( A = (2 2 2 0) \)
- Process 2 can then run: \( A = (4 2 2 1) \)
- Last process can run with no deadlock

When to run deadlock detection algorithms?

- Every resource allocation
- Every \( k \) minutes
- When CPU utilization drops below some threshold
  - CPU will be more likely to be idle if there are many deadlocked processes

Deadlock Recovery

- Once deadlock is detected, we need a way to recover and get the system running again
  - None are particularly attractive

Recovery through Preemption

- In some rare cases, you could manually intervene and suspend a process without destroying computation
  - Printer
    - Collect all sheets already printed
    - Suspend process and give printer to another process
    - Afterward, put sheets back on printer and restart original process
  - This type of recovery is usually impossible and undesirable because it often requires manual intervention

Recovery through rollback

- Design the system to checkpoint processes periodically
- Save memory image, resource state to file
- Multiple checkpoints should be saved to generate a sequence
- When a deadlock is detected, determine which resource and process is causing the problem
  - Roll back execution for that process until before it acquired the resource
  - Force it to wait to reacquire the resource until other previously deadlocked processes are finished with it
- Implications
  - Significant overhead possible
  - Output could potentially have to be discarded
  - Recovery through killing processes
    - Kill a process in the cycle. If that doesn’t solve it, kill another until the cycle is broken
    - Alternative – kill a process outside the cycle that has a resource needed by a process in the cycle.
    - Goal – Try to kill a process that can be run from scratch again (e.g., a compiler)
      - The second run will not be influenced by the partial completion of the first run
- Implications
  - Requires manual intervention generally
  - Simplest solution as well
- Next time – Deadlock Avoidance and Deadlock Prevention