Resource-Allocation Graph (Review)

- The deadlock conditions can be modeled using a directed graph called a resource-allocation graph (RAG)
  - 2 kinds of nodes:
    - Boxes — represent resources
      - Instances of the resource are represented as dots within the box
    - Circles — represent threads / processes
  - 2 kinds of (directed) edges:
    - Request edge — from process to resource
      - indicates the process has requested the resource, and is waiting to acquire it
    - Assignment edge — from resource instance to process — indicates the process is holding the resource instance
  - When a request is made, a request edge is added
    - When request is fulfilled, the request edge is transformed into an assignment edge
    - When process releases the resource, the assignment edge is deleted

Interpreting a RAG With Single Resource Instances (Review)

- If the graph does not contain a cycle, then no deadlock exists
- If the graph does contain a cycle, then a deadlock does exist
- With single resource instances, a cycle is a necessary and sufficient condition for deadlock

Deadlock Detection (Single Resource of Each Type)

- If all resources have only a single instance, deadlock can be detected by searching the resource-allocation graph for cycles
  - Silberschatz defines a simpler graph, called the wait-for graph, and searches that graph instead
    - The wait-for graph is the resource-allocation graph, minus the resources
    - An edge from p1 to p2 means p1 is waiting for a resource that p2 holds (here we don’t care which resource is involved)

One simple algorithm:

- Start at each node, and do a depth-first search from there
- If a search ever comes back to a node it’s already found, then it has found a cycle

Interpreting a RAG With Multiple Resource Instances

- If the graph does not contain a cycle, then no deadlock exists
- If the graph does contain a cycle, then a deadlock may exist
- With multiple resource instances, a cycle is a necessary (but not sufficient) condition for deadlock
Interpreting a RAG With Multiple Resource Instances (cont.)

- If the graph does contain a knot (and a cycle), then a deadlock does exist.

```
               r1
              /  \
            p1   p2
              \
            r3
```

- If the graph does not contain a knot, then a deadlock does not exist.

```
               r1
              /  \
            p1   p2
              \
            r3
```

- With multiple resource instances, a knot is a sufficient condition for deadlock.

Deadlock Detection (Multiple Resources of Each Type)

- This algorithm (Coffman, 1971) uses the following data structures:
  - Existing Resources: (E1, E2, E3, ..., Em)
  - Available Resources: (A1, A2, A3, ..., Am)
  - Current Allocation:
    - C11, C12, C13, ..., C1m
    - C21, C22, C23, ..., C2m
    - Cn1, Cn2, Cn3, ..., Cnm
  - Request:
    - R11, R12, R13, ..., R1m
    - R21, R22, R23, ..., R2m
    - Rn1, Rn2, Rn3, ..., Rnm

- n processes, m types of resources
  - Existing Resources vector tells number of resources of each type that exist
  - Available Resources vector tells number of resources of each type that are available (unassigned to any process)
  - i-th row of Current Allocation matrix tells number of resources of each type allocated (assigned) to process i

Deadlock Detection Algorithm (Multiple Resources of Each Type)

- Every resource is either allocated or available
  - Number of resources of type j that have been allocated to all processes, plus number of resources of type j that are available, should equal number of resources of type j in existence

- Processes may have unfulfilled requests
  - i-th row of Request matrix tells number of resources of each type process i has requested, but not yet received

- Notation: comparing vectors
  - If A and B are vectors, the relation A ≤ B means that each element of A is less than or equal to the corresponding element of B (i.e., A ≤ B iff A_i ≤ B_i for 0 ≤ i ≤ m)
  - Furthermore, A < B iff A ≤ B and A ≠ B

- Operation:
  - Every process is initially unmarked
  - As algorithm progresses, processes will be marked, which indicates they are able to complete, and thus are not deadlocked
  - When algorithm terminates, any unmarked processes are deadlocked

- Algorithm:
  1. Look for an unmarked process Pi for which the i-th row of the Request matrix is less than or equal to the Available vector
  2. If such a process is found, add the i-th row of the Current matrix to the Available vector, mark the process, and go back to step 1
  3. If no such process exists, the algorithm terminates
Deadlock Detection Example
(Multiple Resources of Each Type)

<table>
<thead>
<tr>
<th>Existing Resources</th>
<th>Available Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4 2 3 1)</td>
<td>(2 1 0 0)</td>
</tr>
</tbody>
</table>

Current Allocation

<table>
<thead>
<tr>
<th></th>
<th>Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0</td>
<td>2 0 0 1</td>
</tr>
<tr>
<td>2 0 1</td>
<td></td>
</tr>
<tr>
<td>0 1 2 0</td>
<td>2 1 0 0</td>
</tr>
</tbody>
</table>

resources = (tape drive, plotter, printer, CDROM)

- Whose request can be fulfilled?
  - Process 1 — no — no CDROM available
  - Process 2 — no — no printer available
  - Process 3 — yes — give it the requested resources, and after it completes and releases those resources, A = (2 2 2 0)
  - Process 1 still can’t run (no CDROM), but process 2 can run, giving A = (4 2 2 1)
  - Process 1 can run, giving A = (4 2 3 1)

After Deadlock Detection:
Deadlock Recovery

- How often does deadlock detection run?
  - After every resource request?
  - Less often (e.g., every hour or so, or whenever resource utilization gets low)?

- What if OS detects a deadlock?
  - Terminate a process
    - All deadlocked processes
    - One process at a time until no deadlock
      - Which one?
      - One with most resources?
      - One with less cost?
        » CPU time used, needed in future
        » Resources used, needed
      - That’s a choice similar to CPU scheduling
  - Is it acceptable to terminate process(es)?
    - May have performed a long computation
      » Not ideal, but OK to terminate it
    - Maybe have updated a file or done I/O
      » Can’t just start it over again!

Any less drastic alternatives?

- Preempt resources
  - One at a time until no deadlock
  - Which “victim”?
    - Again, based on cost, similar to CPU scheduling
  - Is rollback possible?
    - Preempt resources — take them away
    - Rollback — “roll” the process back to some safe state, and restart it from there
      » OS must checkpoint the process frequently — write its state to a file
    - Could roll back to beginning, or just enough to break the deadlock
      » This second time through, it has to wait for the resource
      » Has to keep multiple checkpoint files, which adds a lot of overhead
  - Avoid starvation
    - May happen if decision is based on same cost factors each time
    - Don’t keep preempting same process (i.e., set some limit)

After Deadlock Detection:
Deadlock Recovery (cont.)