Dealing with Deadlock (Review)

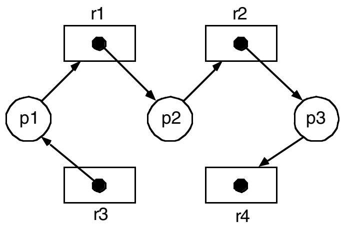
- *The Ostrich Approach* — stick your head in the sand and ignore the problem
- *Deadlock prevention* — prevent deadlock from occurring by eliminating one of the 4 deadlock conditions
- *Deadlock detection* algorithms — detect when deadlock has occurred
  - *Deadlock recovery* algorithms — break the deadlock
- *Deadlock avoidance* algorithms — consider resources currently available, resources allocated to each process, and possible future requests, and only fulfill requests that will not lead to deadlock

Resource-Allocation Graph (Review)

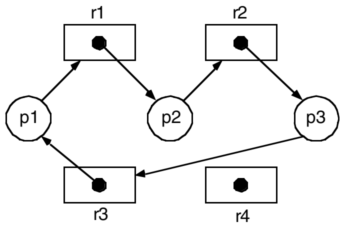
- The deadlock conditions can be modeled using a directed graph called a *resource-allocation graph* (RAG)
  - 2 kinds of nodes:
    - *Boxes* — represent resources
      - Instances of the resource are represented as dots within the box
    - *Circles* — represent processes
  - 2 kinds of (directed) 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 <u>cycle</u>, then **no** deadlock exists



- If the graph **does** contain a <u>cycle</u>, then a deadlock **does** exist



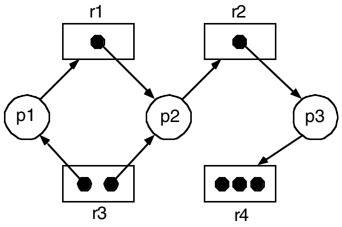
- With **<u>single</u>** resource instances, a **<u>cycle</u>** is a **<u>necessary</u>** and **<u>sufficient</u>** condition for deadlock

Deadlock Detection (Single Resource of Each Type)

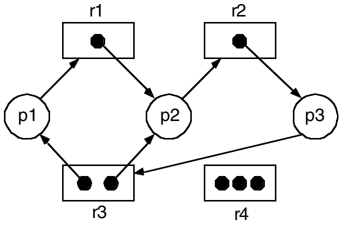
- If all resources have only a single instance, deadlock can be detected by searching the resource-allocation graph for cycles
  - Silberschatz defines a simpler graph, called the *wait-for* graph, and searches that graph instead
    - The wait-for graph is the resource-allocation graph, minus the resources
    - 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 <u>cycle</u>, then **no** deadlock exists



- If the graph **does** contain a <u>cycle</u>, then a deadlock **may** exist

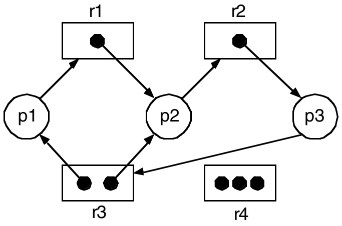


- With **<u>multiple</u>** resource instances, a **<u>cycle</u>** is a **<u>necessary</u>** (but not **<u>sufficient</u>**) condition for deadlock

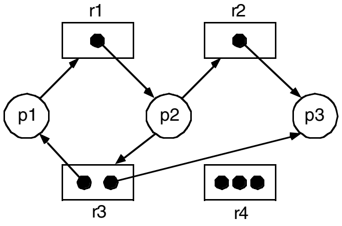
---

Interpreting a RAG With Multiple Resource Instances (cont.)

- If the graph **does** contain a <u>knot</u> (and a cycle), then a deadlock **does** exist



- If the graph **does not** contain a <u>knot</u>, then a deadlock **does not** exist



- With **<u>multiple</u>** resource instances, a **<u>knot</u>** is a **<u>sufficient</u>** 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

$$\begin{bmatrix} C11 & C12 & C13 & \cdots & C1m \\ C21 & C22 & C23 & \cdots & C2m \\ \vdots & \vdots & \vdots & & \vdots \\ Cn1 & Cn2 & Cn3 & \cdots & Cnm \end{bmatrix}$$

Request

$$\begin{bmatrix} R11 & R12 & R13 & \cdots & R1m \\ R21 & R22 & R23 & \cdots & R2m \\ \vdots & \vdots & \vdots & & \vdots \\ Rn1 & Rn2 & Rn3 & \cdots & Rnm \end{bmatrix}$$

- 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 (Multiple Resources of Each Type) (cont.)

- 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 $A \le B$ means that each element of A is less than or equal to the corresponding element of B (i.e., $A \le B$ iff $A_i \le B_i$ for $0 \le i \le m$)
  - Furthermore, $A < B$ iff $A \le B$ and $A \ne B$

Deadlock Detection Algorithm (Multiple Resources of Each Type)

- 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)

Existing Resources (4 2 3 1)

Available Resources (2 1 0 0)

Current Allocation

$$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request

$$\begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

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!

After Deadlock Detection: Deadlock Recovery (cont.)

- 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)