Deadlock Conditions

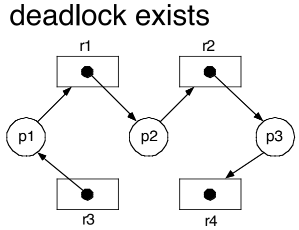
- These 4 conditions are **necessary** and **sufficient** for deadlock to occur:
  - **Mutual exclusion** — if one process holds a resource, other processes requesting that resource must wait until the process releases it (only one can use it at a time)
  - **No preemption** — resources are released voluntarily; neither another process nor the OS can force a process to release a resource
  - **Hold and wait** — processes are allowed to *hold* one (or more) resource and be *waiting* to acquire additional resources that are being held by other processes
  - **Circular wait** — there must exist a set of waiting processes such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, … Pn-1 is waiting for a resource held by Pn, and Pn is waiting for a resource held P0

Resource-Allocation Graph

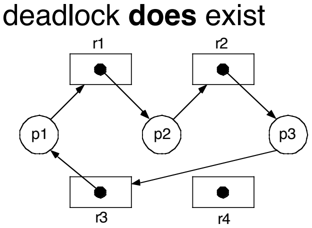
- The deadlock conditions can be modeled using a directed graph called a *resource-allocation graph* (RAG)
  - 2 kinds of nodes:
    - *Boxes* — represent resources
      - Instances of the resource are represented as dots within the box
    - *Circles* — represent processes
  - 2 kinds of (directed) 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

- If the graph does **not** contain a <u>cycle</u>, then **no** deadlock exists



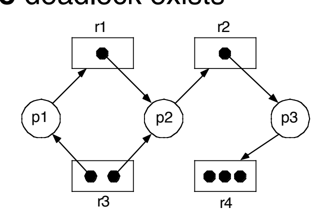
- If the graph **does** contain a <u>cycle</u>, then a deadlock **does** exist



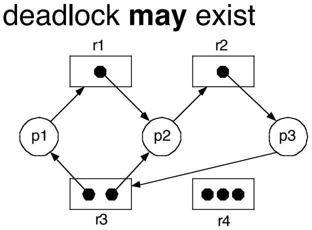
- With **<u>single</u>** resource instances, a **<u>cycle</u>** is a **<u>necessary</u>** and **<u>sufficient</u>** condition for deadlock

Interpreting a RAG With Multiple Resource Instances

- If the graph does **not** contain a <u>cycle</u>, then **no** deadlock exists



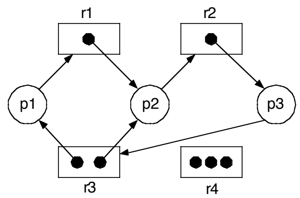
- If the graph **does** contain a <u>cycle</u>, then a deadlock **may** exist



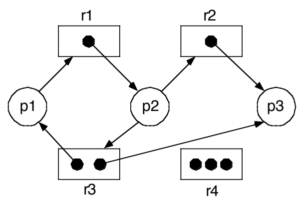
- With **<u>multiple</u>** resource instances, a **<u>cycle</u>** is a **<u>necessary</u>** (but not **<u>sufficient</u>**) condition for deadlock

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

Dealing with Deadlock

- *The Ostrich Approach* — stick your head in the sand and ignore the problem
  - Often used in centralized systems!
  - Maybe also be a good solution for distributed systems in many situations
- *Deadlock avoidance* — consider each resource request, and only fulfill those that will not lead to deadlock
  - Stay in a *safe state* — a state with no deadlock where resource requests can be granted in some order such that all processes will complete
  - ✘ A bad solution for centralized systems, even worse in distributed systems
    - Must know resource requirements of all processes in advance
    - Resource request set is known and fixed, resources are known and fixed
    - Complex analysis for every request

Dealing with Deadlock (cont.)

- *Deadlock prevention* — eliminate one of the 4 deadlock conditions
  - Occasionally used in centralized systems!
  - Maybe also be a good solution for distributed systems in some situations
  - We'll come back to this next time…
- *Deadlock detection* and *recovery* — detect, then break the deadlock
  - Not too hard for single resource instances, harder for multiple resource instances
  - ✘ More difficult when state is distributed
  - ✔ Can detect concurrently w/ other activities

➡ In distributed systems — assume only one non-sharable resource of each type

Deadlock Detection in a Distributed Environment

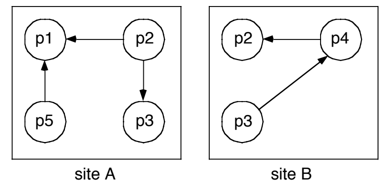
- Centralized algorithms
  - Coordinator maintains global WFG and searches it for cycles
  - Ho and Ramamoorthy's two-phase and one-phase algorithms
- Distributed algorithms
  - Global WFG, with responsibility for detection spread over many sites
  - Obermarck's path-pushing
  - Chandy, Misra, and Haas's edge-chasing
- Hierarchical algorithms
  - Hierarchical organization, site detects deadlocks involving only its descendants
  - Menasce and Muntz's algorithm
  - Ho and Ramamoorthy's algorithm

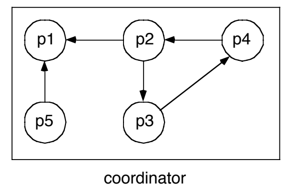
Centralized Deadlock Detection (Simple Algorithms)

- First Algorithm
  - A central coordinator maintains a global wait-for graph (WFG) for the system
    - When appropriate, it checks the WFG for cycles (for single resource instances, a cycle implies deadlock)
    - WFG is resource-allocation graph minus resources; shows that a process is waiting for a resource held by another process
  - All sites request and release resources (even local resources) by sending *request* and *release* messages to the coordinator
    - When coordinator receives a *request*, it
      - updates the global WFG
      - checks for deadlocks
      - grants the request if no deadlock results
    - When coordinator receives a *release*, it
      - updates the global WFG
  - ✘ Large communication overhead, coordinator is a performance bottleneck and single point of failure, etc.

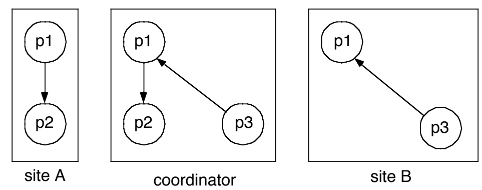
Centralized Deadlock Detection (Example Using Simple Algorithms)

- Cycle in global WFG ⇒ deadlock





- No cycle in global WFG ⇒ no deadlock

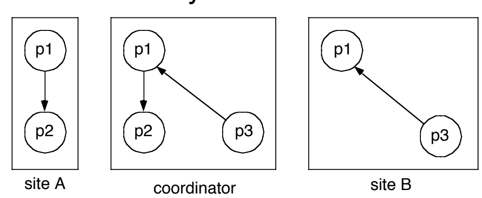


Centralized Deadlock Detection (Simple Algorithms) (cont.)

- Second Algorithm
  - A central coordinator maintains a global wait-for graph (WFG) for the system
    - Individual sites also maintain local WFGs for local processes and resources
    - Global WFG is an <u>approximation</u> of the total state of the system
  - When should the coordinator update the WFG and try to detect deadlocks?
    1. Whenever a new edge is inserted or removed in a local WFG
       - Site informs coordinator via a message
       - Global WFG can be slightly out-of-date
    2. Periodically, when a number of changes have been made to WFG
       - Site sends several changes at once
       - Global WFG can be more out-of-date
    3. Whenever it needs to detect deadlock
  - After deadlock is detected, coordinator selects a "victim", and tells all the sites, which take the appropriate action

Centralized Deadlock Detection (Problem of False Deadlock)

- Consider this system state:



  - Now assume process p2 releases resource p1 is waiting on
  - Slightly thereafter, process p2 requests resource p3 is holding
  - However, first message reaches coordinator after second message
  - The global WFG now has a *false cycle,* which leads to a report of *false deadlock*
- Lamport's algorithm can append logical clock values to each message and avoid this problem, although at the cost of many more messages

Centralized Deadlock Detection (Ho and Ramamoorthy, 1982)

- Two-phase algorithm:
  - Every site maintains a status table, containing status of all local processes
    - Resources held, resources waiting on
  - Periodically, coordinator requests all status tables, builds a WFG, and searches it for cycles
    - No cycles ⇒ no deadlock
    - If cycle is found, coordinator again requests all status tables, again builds a WFG, but this time uses only those edges common to both sets of status tables
  - Rationale was that by using information from two consecutive reports, coordinator would get a consistent view of the state
    - However, it was later shown that a deadlock in this WFG does not imply a deadlock exists
    - So, the HR-two-phase algorithm may reduce the possibility of reporting false deadlocks, but doesn't eliminate it

Centralized Deadlock Detection (Ho and Ramamoorthy) (cont.)

- One-phase algorithm:
  - Every site maintains two status tables
    - *Resource status table* keeps track of processes that are holding or requesting resources at that site
    - *Process status table* keeps track of resources requested or held by processes at that site
  - Periodically, coordinator requests all status tables, builds a WFG using only information in both a resource and process table, and searches it for cycles
  - Rationale was that this eliminates inconsistency caused by network delay
    - Message in transit will have entry at one site, not yet at the other
    - ✓ The HR-one-phase algorithm does not report false deadlocks
  - Compared to two-phase algorithm:
    - ✓ Faster, less messages
    - ✗ More storage (2 tables), bigger messages