Dealing with Deadlock (Review)

- The Ostrich Approach — stick your head in the sand and ignore the problem
- Deadlock avoidance — consider resources and requests, and only fulfill requests that will not lead to deadlock
  - Too hard for centralized systems, even harder in distributed systems!!
- Deadlock prevention — eliminate one of the 4 deadlock conditions
- Deadlock detection and recovery — detect, then break the deadlock
  - More difficult when state is distributed
  - Must avoid reporting false deadlock
  - In distributed systems, we typically assume single resource instances

Deadlock Detection in a Distributed Environment (Review)

- 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

Distributed Deadlock Detection

- Path-pushing
  - WFG is disseminated as paths — sequences of edges
  - Deadlock if process detects local cycle
- Edge-chasing
  - Probe messages circulate
  - Blocked processes forward probe to processes holding requested resources
  - Deadlock if initiator receives own probe
- Diffusion
  - Query messages sent to dependent set
  - Active processes discard query, blocked processes forward query under certain conditions, reply under other conditions
  - Deadlock if initiator receives replies to all its queries

Distributed Deadlock Detection (Obermarck’s Path-Pushing, 1982)

- Individual sites maintain local WFGs
  - Nodes for local processes
  - Node “Pex” represents external processes
- Deadlock detection:
  - If a site Si finds a cycle that does not involve Pex, it has found a deadlock
  - If a site Si finds a cycle that does involve Pex, there is the possibility of a deadlock
    - It sends a message containing its detected cycle to any sites involved in Pex
    - If site Sj receives such a message, it updates its local WFG graph, and searches it for a cycle
      - If Sj finds a cycle that does not involve its Pex, it has found a deadlock
      - If Sj finds a cycle that does involve its Pex, it sends out a message…
  - Can report false deadlock
Distributed Deadlock Detection (Obermarck’s Path-Pushing) (cont.)

- **Example:**
  
  Initial state:

  Site A detects cycle, sends message describing that cycle to Site B:

  Site B updates its WFG, finds cycle not involving Pex \( \Rightarrow \) deadlock

Distributed Deadlock Detection (Chandy, Misra, and Haas’s Edge-Chasing, 1983)

- When a process has to wait for a resource (blocks), it sends a *probe* message to process holding the resource:
  - Process can request (and can have to wait for) multiple resources at once
  - *Probe* message contains 3 values:
    - ID of process that blocked
    - ID of process sending message
    - ID of process message was sent to

- When a blocked process receives a probe, it propagates the probe to the process(es) holding resources that it has requested:
  - ID of blocked process stays the same, other two values updated as appropriate
  - If the blocked process receives its own probe, there is a deadlock

Distributed Deadlock Detection (Chandy, Misra, and Haas’s Edge-Chasing) (cont.)

- **Example where p1 initiates deadlock detection by sending a probe:**

  ✔ Doesn’t report false deadlock (why not?)

  ✔ Easy to implement, small messages, relatively small number of messages

  ✔ Don’t have to collect and maintain WFGs

Distributed Deadlock Detection (Evaluation of Algorithms)

- **Distributed deadlock detection**
  - Sites share responsibility for WFG and deadlock detection
  - ✔ No single point of failure
  - ✔ Robust — multiple sites can detect the same deadlock
  - ✗ Avoiding false deadlock is hard

- **Obermarck’s path-pushing**
  - \( n(n-1)/2 \) messages to detect deadlock
    - \( n \) sites
    - size of a message is \( O(n) \)

- **Chandy, Misra, and Haas’s edge chasing:**
  - \( m(n-1)/2 \) messages to detect deadlock
    - \( m \) processes, \( n \) sites
    - size of a message is 3 integers
Hierarchical Deadlock Detection

- Sites are organized hierarchically
  - A site is only responsible for detecting deadlocks involving its children sites

- Menasce and Muntz, 1979
  - Sites (called controllers) are organized as a tree
    - Leaf controllers manage resources
      - Each maintains a local WFG concerned only about its own resources
    - Interior controllers are responsible for deadlock detection
      - Each maintains a global WFG that is the union of the WFGs of its children
      - Detects deadlock among its children
  - Whenever a controller changes its WFG due to a resource request, it propagates that change to its parent
    - Parent updates its WFG, and searches it for cycles, propagates changes upward

Hierarchical Deadlock Detection (cont.)

- Ho and Ramamoorthy, 1982
  - Sites are grouped into disjoint clusters
  - Periodically, a site is chosen as a central control site
    - Central control site chooses a control site for each cluster
  - Control site collects status tables from its cluster, and uses the Ho and Ramamoorthy one-phase centralized deadlock detection algorithm to detect deadlock in that cluster
  - All control sites then forward their status information and WFGs to the central control site, which combines that information into a global WFG and searches it for cycles
  - Control sites detect deadlock in clusters
  - Central control site detects deadlock between clusters

Perspective

- Correctness of algorithms
  - There are few formal methods to prove the correctness of deadlock detection algorithms — we usually use informal or intuitive arguments

- Performance
  - Usually measured as the number of messages exchanged to detect deadlock
    - Deceptive since message are also exchanged when there is no deadlock
    - Doesn’t account for size of the message
  - Should also measure:
    - Deadlock persistence time (measure of how long resources are wasted)
      - Tradeoff with communication overhead
    - Storage overhead (graphs, tables, etc.)
    - Processing overhead to search for cycles
    - Time to optimally recover from deadlock

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)