Distributed Systems

14. Distributed Deadlock

Paul Krzyzanowski
Rutgers University
Fall 2015
Deadlocks

Four conditions for deadlock

1. Mutual exclusion
2. Hold and wait
3. Non-preemption
4. Circular wait
Deadlock

- Resource allocation
  - Resource $R_1$ is allocated to process $P_1$
    
    ![Diagram](Diagram)
    
    $P_1 \xrightarrow{holds} R_1$

  - Resource $R_1$ is requested by process $P_1$

    ![Diagram](Diagram)
    
    $R_1 \xleftarrow{wants} P_1$

- Deadlock is present when the graph has cycles

- This graph is called a **Wait-For Graph (WFG)**
Deadlock example

Circular dependency among four processes and four resources leads to deadlock
Dealing with deadlock

- Same conditions for distributed systems as centralized
- Harder to detect, avoid, prevent

Strategies

1. Ignore
   - Do nothing. So easy. So tempting.

2. Detect
   - Allow the deadlock to occur, detect it, and then deal with it by aborting and restarting a transaction that causes deadlock

3. Prevent
   - Make deadlock impossible by granting requests such that one of the conditions necessary for deadlock does not hold

4. Avoid
   - Choose resource allocation so deadlock does not occur (but algorithm needs to know what resources will be used and when)
Deadlock detection

• Kill off one or more processes when deadlock is detected
  – That breaks the circular dependency
  – It also doesn’t feel like a great thing to do

• But transactions are designed to be abortable

• Just abort one or more transactions
  – System restored to state before transaction began
  – Transaction can restart at a later time
  – Resource allocation in system may be different then so the transaction may succeed
Centralized deadlock detection

• Imitate the non-distributed algorithm through a coordinator

• Each machine maintains a **Wait-For Graph** for its processes and resources

• A central coordinator maintains the combined graph for the entire system: the **Global Wait-For Graph**
  – A message is sent to the coordinator each time an edge (resource hold/request) is added or deleted
  – List of adds/deletes can be sent periodically
Centralized deadlock detection

Local Wait-For Graph on A

Global Wait-For Graph

Local Wait-For Graph on B
Two events occur:
1. Process $P_1$ releases resource $R$ on system $A$
2. Process $P_1$ asks system $B$ for resource $T$

Two messages are sent to the coordinator:
1. (from $A$): $release\ R$
2. (from $B$): $wait\ for\ T$

If message 2 arrives first, the coordinator constructs a graph that has a cycle and hence detects a deadlock. This is **false deadlock**.

Globally consistent (total) ordering must be imposed on all processes or Coordinator can reliably ask each process whether it has any release messages.

A **false deadlock** is sometimes known as a **phantom deadlock**.
False Deadlock Example

No deadlock

P₀: release(R)

Still no deadlock!

P₁: wait_for(T)
False Deadlock Example

We detected deadlock because the coordinator received the messages out of order.
Avoiding False Deadlock

Impose globally consistent (total) ordering on all processes

or

Have coordinator reliably ask each process whether it has any release messages
Distributed deadlock detection

• Processes can request multiple resources at once
  – Consequence: process may wait on multiple resources

• Some processes wait for local resources

• Some processes wait for resources on other machines

• Algorithm invoked when a process has to wait for a resource
Distributed detection algorithm

Chandy-Misra-Haas algorithm

**Edge Chasing**

**Probe** message is generated

- Sent to all process(es) holding the needed resources
- Message contains three process IDs: \{blocked ID, my ID, holder ID\}
  1. Process that just blocked
  2. Process sending the message
  3. Process to whom the message is being sent
Distributed detection algorithm

• When *probe* message arrives, recipient checks to see if it is waiting for any processes
  – if so, update & forward message: \{blocked ID, my ID, holder ID\}
    • Replace second field by its own process number
    • Replace third field by the number of the process it is waiting for
    • Send messages to each process on which it is blocked

• If a message goes all the way around and comes back to the original sender, a cycle exists
  – *We have deadlock*
Distributed deadlock detection

- Process 0 is blocking on process 1
  - initial message from 0 to 1: (0,0,1)
  - $P_1$ sends (0, 1, 2) to $P_2$; $P_2$ sends (0, 2, 3) to $P_3$
- Message (0,8,0) returns back to sender
  - cycle exists: deadlock
Distributed deadlock prevention

• Design system so that deadlocks are structurally impossible

• Disallow at least one of conditions for deadlock
  – Mutual exclusion
    • Allow a resource to be held (used) by more than one process at a time. Not practical if an object gets modified.
  – Hold and wait
    • Implies that a process gets all of its resources at once. Not practical to disallow this – we don’t know what resources a process will use.
  – Non-preemption
    • This can violate the ACID properties of a transaction. We can use optimistic concurrency control algorithms and check for conflicts at commit time and roll back if needed
  – Circular wait
    • Ensure that a cycle of waiting on resources does not occur.
Distributed deadlock prevention

• Deny circular wait
• Assign a unique timestamp to each transaction
• Ensure that the *Global Wait-For Graph* can only proceed from *young to old* or from *old to young*
Deadlock prevention

- When a process is about to block waiting for a resource used by another
  - Check to see which has a larger timestamp (which is older)
- Allow the wait only if the waiting process has an older timestamp (is older) then the process waited for
- Following the resource allocation graph, we see that timestamps always have to increase, so cycles are impossible.
- Alternatively: allow processes to wait only if the waiting process has a higher (younger) timestamp than the process waiting for.
Wait-die algorithm

- Old process wants resource held by a younger process
  - old process waits

- Young process wants resource held by older process
  - young process kills itself

This is the **wait-die** algorithm
• Instead of killing the transaction making the request, kill the resource owner

• Old process wants resource held by a younger process
  – old process kills the younger process

• Young process wants resource held by older process
  – young process waits

This is the **wound-wait** algorithm
The End