Distributed Systems
13. Distributed Deadlock

Paul Krzyzanowski
Rutgers University
Fall 2018

October 22, 2018
© 2014 - 2018 Paul Krzyzanowski

Deadlock

Four conditions for deadlock
1. Mutual exclusion
2. Hold and wait
3. Non-preemption
4. Circular wait

October 22, 2018
© 2014 - 2018 Paul Krzyzanowski

Deadlock

Resource allocation
- Resource R₁ is allocated to process P₁
  \[ P₁ \text{ holds } R₁ \]
- Resource R₁ is requested by process P₁
  \[ R₁ \text{ wants } P₁ \]

This graph is called a Wait-For Graph (WFG)
Deadlock is present when the graph has cycles

October 22, 2018
© 2014 - 2018 Paul Krzyzanowski

Wait-For Graph: Deadlock Example

Circular dependency among four processes and four resources leads to deadlock

October 22, 2018
© 2014 - 2018 Paul Krzyzanowski

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 the algorithm needs to know what resources will be used and when (not feasible in most cases)

October 22, 2018
© 2014 - 2018 Paul Krzyzanowski

Deadlock detection

- Kill off one or more processes when deadlock is detected
  - That breaks the circular dependency
- It might not feel good to kill a process
  - But if we’re using transactions, transactions are designed to be abortable
- So just abort one or more transactions
  - System restored to state before transaction began
  - Transaction can restart at a later time
  - Resource allocation in the system may be different then so the transaction may succeed

October 22, 2018
© 2014 - 2018 Paul Krzyzanowski
Centralized deadlock detection

- Imitate the non-distributed algorithm through a coordinator
- Each system 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

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. \( (A) \): release \( R \)
2. \( (B) \): wait for \( T \)

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

Phantom Deadlock Example

- No deadlock
- Message 1 from \( P_1 \): release(\( R \))
- Message 2 from \( P_1 \): wait_for(\( T \))

All good: no deadlock detected!

Avoiding Phantom Deadlock

- Impose globally consistent (total) ordering on all processes
- Have coordinator reliably ask each process whether it has any release messages

We detected deadlock because the coordinator received the messages out of order
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**
When requesting a resource, generate a probe message
- Send to all process(es) currently holding the needed resource
- Message contains three process IDs: \((\text{blocked\_ID, my\_ID, holder\_ID})\)
  1. Process that originated the message \((\text{blocked\_ID})\)
  2. Process sending (or forwarding) the message \((\text{my\_ID})\)
  3. Process to whom the message is being sent \((\text{holder\_ID})\)

Distributed deadlock prevention

Design system so that deadlocks are structurally impossible

- **Mutual exclusion**
  - Allow a resource to be held (used) by more than one process at a time.
  - Not practical if an object gets modified.
  - This can violate the ACID properties of a transaction
- **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**
  - Essentially gives up mutual exclusion.
  - This can also 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 detection

- Process 0 needs a resource process 1 is holding
  - That means process 0 will block on process 1
    - Initial message from P\(_0\) to P\(_1\): \((0, 0, 1)\)
    - 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
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

October 22, 2018
© 2014-2018 Paul Krzyzanowski

Wound-wait 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

Only permit younger processes to wait on resources held by older processes.

The end