Distributed deadlocks

CS 417

Distributed Systems

Deadlocks

Four conditions

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

A *deadlock* is a condition where a process cannot proceed because it needs to obtain a resource held by another process and it itself is holding a resource that the other process needs.

We can consider two types of deadlock:

communication deadlock occurs when process A is trying to send a message to process B, which is trying to send a message to process C which is trying to send a message to A.

A **resource deadlock** occurs when processes are trying to get exclusive access to devices, files, locks, servers, or other resources. We will not differentiate between these types of deadlock since we can consider communication channels to be resources without loss of generality.

Four conditions have to be met for deadlock to be present:

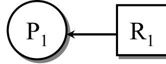
1. **Mutual exclusion.** A resource can be held by at most one process
2. **Hold and wait.** Processes that already hold resources can wait for another resource.
3. **Non-preemption.** A resource, once granted, cannot be taken away from a process.
4. **Circular wait.** Two or more processes are waiting for resources geld by one of the other processes.

We can represent resource allocation as a graph where: **P ←R** means a resource R is currently held by a process P. **P→R** means that a process P wants to gain exclusive access to resource R.

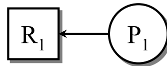
Deadlock exists when a resource allocation graph has a cycle.

Deadlocks

- Resource allocation
  - Resource R_1 is allocated to process P_1



  - Resource R_1 is requested by process P_1



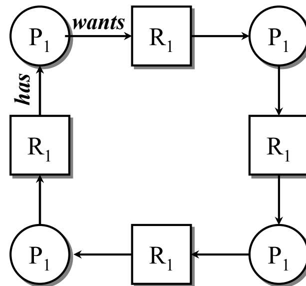
- Deadlock is present when the graph has cycles

We can represent resource allocation as a graph where: **P → R** means a resource R is currently held by a process P.

R → P means that a process P wants to gain exclusive access to resource R.

Deadlock exists when a resource allocation graph has a cycle.

Deadlock example



- Circular dependency among four processes and four resources

The figure illustrates a deadlock condition between 4 processes P1,P2,P3,P4 and four resources: R1, R2, R3, R4.

Process P1 is holding resource R1 and wants resource R3.

Resource R3 is held by process P3 which wants resource R2.

Resource R2 is held by process P2 which wants resource R1.

Resource R1 is held by process P1 and hence we have deadlock.

Deadlocks in distributed systems

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

Deadlocks in distributed systems are similar to deadlocks in centralized systems. In centralized systems, we have one operating system that can oversee resource allocation and know whether deadlocks are (or will be) present. With distributed processes and resources it becomes harder to detect, avoid, and prevent deadlocks.

Several strategies can be used to handle deadlocks:

ignore: we can ignore the problem. This is one of the most popular solutions.

detect: we can allow deadlocks to occur, then detect that we have a deadlock in the system, and then deal with the deadlock

prevent: we can place constraints on resource allocation to make deadlocks impossible

avoid: we can choose resource allocation carefully and make deadlocks impossible. Deadlock avoidance is **never** used (either in distributed or centralized systems). The problem with deadlock avoidance is that the algorithm will need to know resource usage requirements in advance so as to schedule them properly.

Deadlock detection

Preventing or avoiding deadlocks can be difficult.

- Detecting them is easier.
- When deadlock is detected
  - kill off one or more processes
    - annoyed users
  - if system is based on atomic transactions, abort one or more transactions
    - transactions have been designed to withstand being aborted
    - system restored to state before transaction began
    - transaction can start a second time
    - resource allocation in system may be different so the transaction may succeed

General methods for preventing or avoiding deadlocks can be difficult to find.

Detecting a deadlock condition is generally easier.

When a deadlock is detected, it has to be broken. This is traditionally done by killing one or more processes that contribute to the deadlock. Unfortunately, this can lead to annoyed users.

When a deadlock is detected in a system that is based on atomic transactions, it is resolved by aborting one or more transactions. But transactions have been designed to withstand being aborted.

When a transaction is aborted due to deadlock:

- system is restored to the state it had before the transaction began
- transaction can start again
- hopefully, the resource allocation/utilization will be different now so the transaction can succeed

Consequences of killing a process in a transactional system are less severe.

Centralized deadlock detection

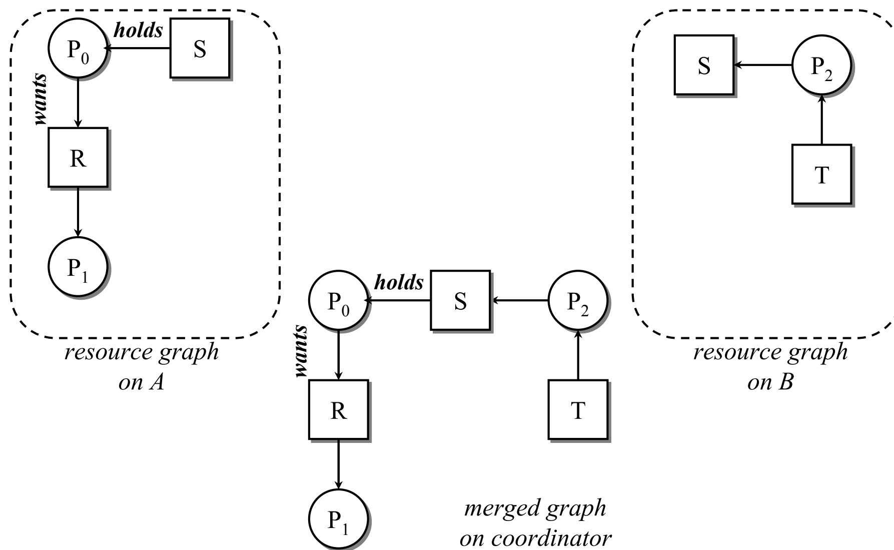
- Imitate the nondistributed algorithm through a coordinator
- Each machine maintains a resource graph for its processes and resources
- A **central coordinator** maintains a graph for the entire system
  - message can be sent to coordinator each time an arc is added or deleted
  - list of arc adds/deletes can be sent periodically

The centralized algorithm attempts to imitate the nondistributed algorithm by using a centralized coordinator. Each machine is responsible for maintaining its own processes and resources. The coordinator maintains the resource utilization graph for the entire system.

To accomplish this, the individual subgraphs need to be propagated to the coordinator. This can be done by sending a message each time an arc is added or deleted.

If optimization is needed (reduce # messages) then a list of added or deleted arcs can be sent periodically.

Centralized deadlock detection



Suppose machine *A* has a process P0 which holds resource S and wants resource R. Resource R is held by P1. This local graph is maintained on machine A.

Suppose that another machine B, has a process P2, which is holding resource T and wants resource S.

Both of these machines send their graphs to the coordinator, which maintains the union (overall graph). The coordinator sees no cycles. Therefore there are no deadlocks.

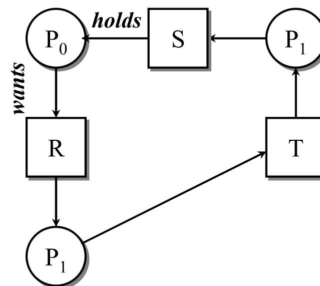
If a cycle was found (hence a deadlock), the coordinator would have to make a decision on which machine to notify for killing a process to break the deadlock.

Centralized deadlock detection

Two events occur:
1. Process P_1 releases resource R
2. Process P_1 asks machine B for resource T

Two messages are sent to the coordinator:
1 (from A): *releasing R*
2 (from B): *waiting 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.**



false deadlock

Global time ordering must be imposed on all machines
or
Coordinator can reliably ask each machine whether it has any release messages.

Suppose two events occur: process P1 releases resource R and asks machine B for resource T. Two messages are sent to the coordinator:

message 1 (from machine A): *releasing R*

message 2 (from machine B): *waiting for T*

This should cause no problems (no deadlock) as no cycles will exist.

However, suppose message 2 arrives first. The coordinator would then construct the graph shown and detect a deadlock (cycle). This condition is known as a **false deadlock**.

A way to fix this is to use Lamport's algorithm to impose global time ordering on all machines.

Alternatively, if the coordinator suspects deadlock, it can send a reliable message to every machine asking whether it has any release messages. Each machine will then respond with either a release message or a message indicating that it is not releasing any resources.

Distributed deadlock detection

- Chandy-Misra-Haas algorithm
- Processes can requests multiple resources at once
  - growing phase of a transaction can be sped up
  - 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

The Chandy-Misra-Haas algorithm is a distributed approach to deadlock detection. The algorithm was designed to allow processes to make requests for multiple resources at once. One benefit of this is that, for transactions, the growing phase of a transaction (acquisition of resources - we'll cover this later) can be sped up. One consequence of this is that a process may be blocked waiting on multiple resources.

Some resources may be local and some may be remote.The cross-machine arcs is what makes deadlock detection difficult.

The algorithm is invoked when a process has to wait for some resource.

Distributed detection algorithm

- **Probe** message is generated
  - sent to process(es) holding the needed resources
  - message contains three numbers
    - process that just blocked
    - process sending the message
    - process to whom it is being sent

The algorithm begins by sending a **probe** message to the proces(es) holding the needed resources. The probe message consists of three numbers:

1. The process that just blocked
2. The process sending the message (initially the same as 1)
3. The process to whom it is being sent.

Distributed detection algorithm

- when ***probe*** message arrives, recipient checks to see if it is waiting for any processes
  - if so, update message
    - 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*

When the *probe* message arrives, the recipient checks to see if it is waiting for any processes. If so, the message is updated:

- the first field remains the same

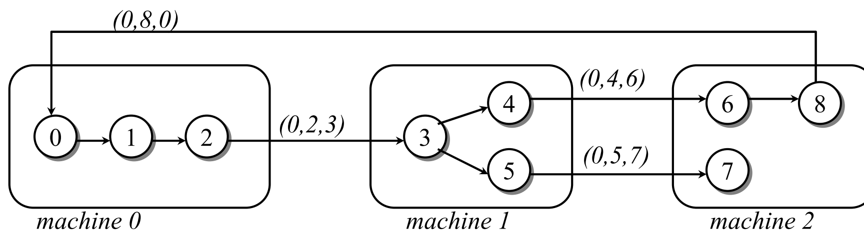
- the second field is replaced by the recipient's process number

- the third field is replaced with the number of the process it is waiting for

If the process is blocked on multiple processes, it sends out multiple messages (with the third field modified appropriately).

If a message goes all the way around and comes back to the original sender (the process listed in the first field) then a cycle exists and the system is deadlocked.

Distributed deadlock detection



- Process 0 is blocking on process 1
  - initial message from 0 to 1: *(0,0,1)*
- Message *(0,8,0)* returns back to sender
  - cycle exists: *deadlock*

The sample resource graph here shows only processes. Each arc passes through a resource, but resources have been omitted for simplicity.

Some processes, such as 0, 1, 3, 6 are waiting for local resources.

Others, such as 2, 4, 5, 8 are waiting for resources on other machines.

Suppose process 0 requests a resource held by process 1. It constructs a **probe** message containing (0, 0, 1) and sends it to process 1. Process 1 changes the probe to (0,1, 2) and sends it to the process whose resources it is holding: 2. Process 2, in turn, constructs a probe to a remote process 3. Process 3 is blocking on two resources, so it has to send two probes out. Process 4 gets (0,3,4) and process 5 gets (0,3,5). Eventually process 8 gets the probe and sends it out to the process that it's blocking on: 0. When 0 receives the probe and sees that the first element is its own process ID, it knows that a cycle has been detected and the system is deadlocked.

Distributed deadlock prevention

- Design system so that deadlocks are structurally impossible
- Various techniques exist:
  - allow processes to hold one resource at a time
  - require all processes to request all resources initially and release them all when asking for a new one
  - order all resources and require processes to acquire them in increasing order (making cycles impossible)
- With global time and atomic transactions: two other techniques
  - based on idea of assigning each transaction a global timestamp when it starts

Deadlock prevention algorithms deal with designing the system in such a way that deadlocks cannot occur.

There are a number of techniques that exist to accomplish this:

> allow processes to hold one resource at a time
>
> require all processes to request all resources initially and release them all when asking for a new one
>
> order all resources and require processes to acquire them in increasing order (making cycles impossible)

These are all rather cumbersome in practice. If we have a distributed system with global time and atomic transactions, two other algorithms are possible. These are based on the idea of assigning a global timestamp to each transaction the moment it starts (no two transactions can have the exact same time stamp -- use Lamport's algorithm or a global sequence number generator.

Deadlock prevention

- When one 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.

When a process is about to block waiting for a resource that another process is using, a comparison is made of the timestamps of the two processes.

We can then allow the wait *only* if the waiting process is older than the process waited for. If we abide by this rule, any resource allocation graph will only have timestamps that increase, so cycles (and hence deadlock) are impossible.

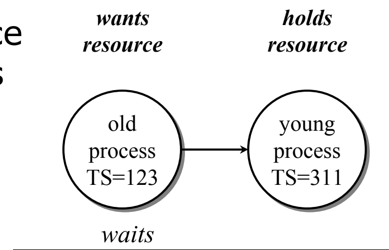
Alternatively: we can allow processes to wait only if the waiting process is younger than the process waited for - timestamps now decrease as we follow the graph and cycles are again impossible.

It is generally preferable to give priority to older processes (they have run longer and are likely to hold more resources -- it also eliminates starvation).

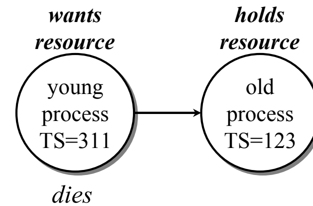
Killing a transaction is relatively harmless since it can be restarted later (with a new timestamp).

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

Suppose an old process (timestamp=123) wants a resource held by a young process (timestamp=311). We don't want to kill off the old process, since this is inefficient, so the old process will wait for the young one to finish using the resource.

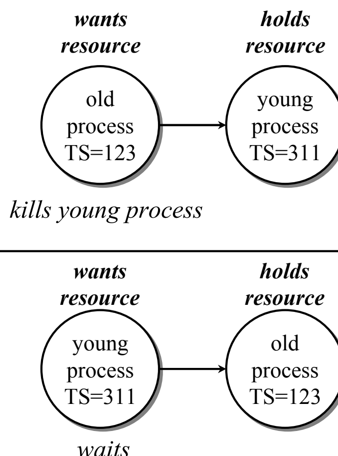
Now suppose that a young process wants a resource held by an older process. In this case, the young process will kill itself.

This assures us that the arrows of resource utilization arcs always point in the direction of increasing transaction numbers, making cycles impossible.

This algorithm is called **wait-die.**

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



This is the **wound-wait** algorithm

If we assume that we have a transactional system, we can now kill the resource owner instead of ourselves. Without transactions, this may have bad consequences (processes may have modified files, etc). With transactions, we know that the act of killing the transaction will undo anything that the transaction has done thus far.

Suppose an old process (timestamp=123) wants a resource held by a young process (timestamp=311). We don't want to kill off the old process, since this is inefficient. Instead of having the old resource wait, as it did for the wait-die algorithm, it will kill the younger transaction.

Now suppose that a young process wants a resource held by an older process. Instead of killing itself, this time it will wait for the old transaction to finish.

This assures us that the arrows of resource utilization arcs always point in the direction of *decreasing* transaction numbers, making cycles impossible.

This algorithm is called **wound-wait.**

The attraction that this has over the wait-die algorithm is that it prevents a young process that wants a resource killing itself, restarting, seeing the resource is still unavailable, killing itself, restarting, and on and on and on...