Distributed System

DeadLock

Classification

- Deadlocks in DS are similar to deadlocks in processor systems, only worse.
- -They are harder to avoid, prevent, or even detect, and harder to cure when tracked down because all the relevant information is scattered over many machines.

  -In distributed data base systems, they can be extremely serious
- Some people make a distinction between two kinds of distributed deadlocks:

  1. communication deadlocks

  - Occurs, for example, when process A is trying to send a message to process B, which in turn is trying to send one to process C, which is trying to send one to A.

  -There are various scenarios in which this situation leads to deadlock, such as no buffers being available.

  2. resource deadlocks.

  -Occurs when processes are fighting over exclusive access to I/O devices, files, locks, or other resources.

Look only to resource deadlocks

- Communication channels, buffers, and so on, are also resources and can be modeled as resource

  -processes can request them and release them.
- Circular communication patterns of the type just described are quite rare in most systs:

  -Example: client-server systems,

  - a client might send a message (or perform an RPC) with a file

server, which might send a message to a disk server.

- it is unlikely that the disk server, acting as a client, would send a message to the original client, expecting it to act like a server.

- Circular wait condition is unlikely to occur as a result of communication alone.

Strategies are used to handle deadlocks

1. The ostrich algorithm (ignore the problem):

-popular in DSs as it is in single-processor systems.

- no system-wide deadlock mechanism is present in DS used for programming, office automation, process control, and many other application

2. Detection (let deadlocks occur, detect them, and try to recover).

-also popular, primarily because prevention and avoidance are so difficult.

3. Prevention (statically make deadlocks structurally impossible).

-more difficult than in uni-processor systems.

4. Avoidance (avoid deadlocks by allocating resources carefully).

-never used in DS

- the problem is that the proposed algorithms need to know (in advance)

how much of each resource every process will eventually need – this information is rarely, if ever, available.

Distributed Deadlock Detection

- When a deadlock is detected in a conventional OS, the way to resolve

it is to kill off one or more processes.

- Doing so invariably leads to one or more unhappy users.

- When a deadlock is detected in a system 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 because it contributes to a deadlock,

= the system is first restored to the state it had before the transaction began,

= at which point the transaction can start again.

=It is probable that it will succeed the second time.

-> The difference is that the consequences of killing off a process are much less severe when transactions are used than when they are not used.

Centralized Deadlock Detection

- Try to imitate the non-distributed algorithm.
- Each machine maintains the resource graph for its own processes and resources,
- A central coordinator maintains the resource graph for the entire system (the union of all the individual graphs).
- Variants:

1. whenever an arc is added or deleted from the resource graph, a message can be sent to the coordinator providing the update.
2. periodically, every process can send a list of arcs added or deleted since the previous update – requires fewer messages than the first one.
3. the coordinator can ask for information when it needs it.

- When the coordinator detects a cycle, it kills off one process to break the deadlock.

False deadlock

- Consider a system with processes A and B running on machine 0, and process C on machine 1.
- Three resources exist: R, S and T.
- Initially:

  - A holds S but wants R, which it cannot have because B is using it;

  - C has T and wants S, too.

  - The coordinator's view of the world is shown in (c)

  - This configuration is safe: as soon as B finishes, A can get R and finish, releasing S for C.
- After a while:

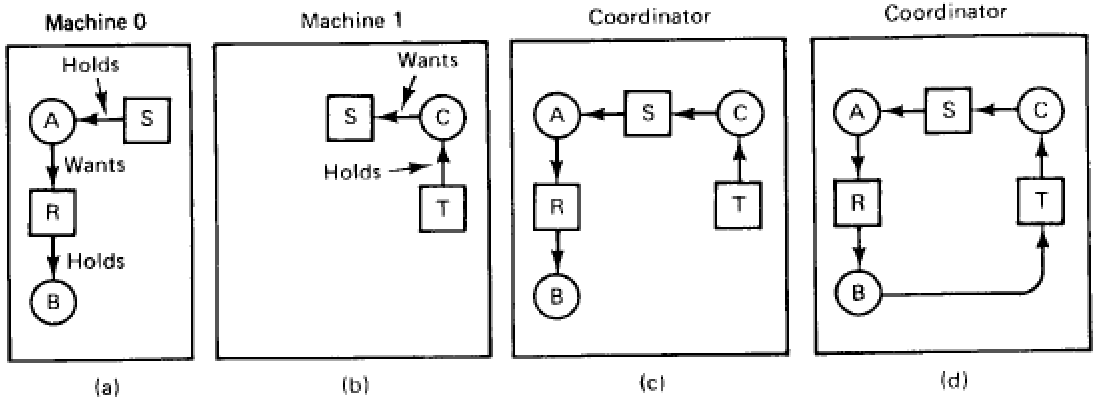
  - B releases R and asks for T, a perfectly legal and safe swap.

  - Machine 0 sends a message to the coordinator announcing the release of R,

  - Machine 1 sends a message to the coordinator announcing the fact that B is now waiting for its resource, T.

  - Unfortunately, the message from machine 1 arrives first, leading the coordinator to construct the graph of (d).

  -The coordinator incorrectly concludes that a deadlock exists and kills some process.
- Such a situation is called a *false deadlock.*


Machine 0
Holds
Wants
Holds
A
S
R
B
Machine 1
Wants
Holds
S
C
T
Coordinator
A
S
C
R
T
B
Coordinator
A
S
C
R
T
B
(a)
(b)
(c)
(d)

Way out: use Lamport's alg.

- Since the message from machine 1 to the coordinator is triggered by the request from machine 0, the message from machine 1 to the coordinator will have a later timestamp than the message from machine 0 to the coordinator.
- When the coordinator gets the message from machine 1 that leads it to

suspect deadlock,

- send a message to every machine in the system saying:

"I just received a message with timestamp T which leads to deadlock.

If anyone has a message for me with an earlier timestamp, please

send it immediately."

- When every machine has replied, positively or negatively, the coordinator will see that the arc from R to B has vanished, so the system is still safe.
- Although this method eliminates the false deadlock, it requires global time and is expensive

Distrib.Deadlock Detection: Chandy-Misra-Haas alg.

- processes are allowed to request multiple resources (e.g. locks)

at once, instead of one at a time.

- by allowing multiple requests simultaneously, the growing phase of a

transaction can be speeded up considerably.

- the consequence of this change to the model is that a process may

now wait on two or more resources simultaneously.

- Example:

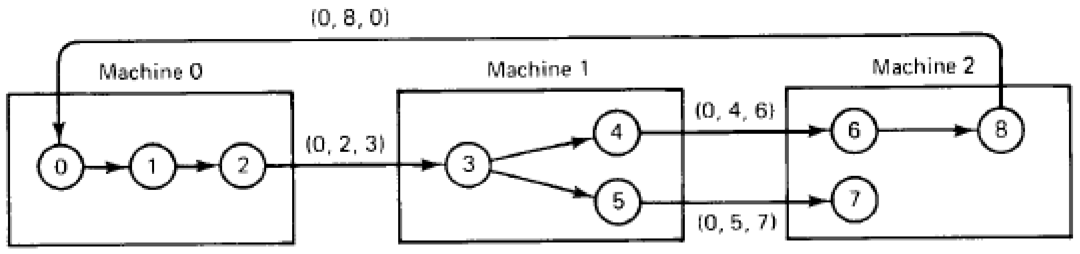
- a modified resource graph, where only the processes are shown

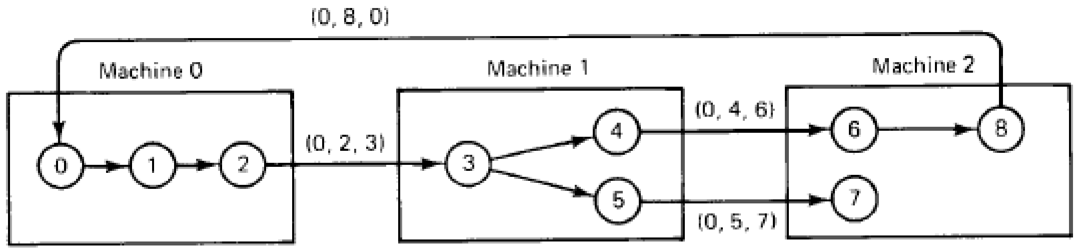
-each arc passes through a resource,

-for simplicity the resources have been omitted from the figure

-process 3 on machine 1 is waiting for two resources, one held by

process 4 and one held by process 5.


(0, 8, 0)
Machine 0
Machine 1
Machine 2
(0, 4, 6)
(0, 2, 3)
(0, 5, 7)
0
1
2
3
4
5
6
7
8


(0, 8, 0)
Machine 0
Machine 1
Machine 2
(0, 4, 6)
(0, 2, 3)
(0, 5, 7)
0
1
2
3
4
5
6
7
8

Example

- Some of the processes are waiting for local resources, such as process 1, but others, such are process 2, are waiting for resources that are located on a different machine.

  -These cross-machine arcs that make looking for cycles difficult.

- Alg. invoked when a process has to wait for some resource, for example, process 0 blocking on process 1.

  -A special probe message is generated and sent to the process (or processes) holding the needed resources.

  - The message consists of three numbers: the process that just blocked, the process sending the message, and the process to whom it is being sent.

  -The initial message from to 1 contains the triple (0, 0, 1).

  - When the message arrives, the recipient checks to see if it itself is waiting for any processes.

  - If so, the message is updated, keeping the first field but replacing the second field by its own process number and the third one by the number of the process it is waiting for.

  - The message is then sent to the process on which it is blocked.

  - If it is blocked on multiple processes, all of them are sent (different) messages.

  - Remote messages labeled (0, 2, 3), (0, 4, 6), (0, 5, 7), and (0, 8, 0).

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

Broken the deadlock from the example

1. One way is to have the process that initiated the probe commit suicide.

- This method has problems if several processes invoke the algorithm simultaneously.

-Imagine that both 0 and 6 block at the same moment, and both

initiate probes.

-Each would eventually discover the deadlock, and each would kill

itself – overkill !

2. An alternative algorithm is to have each process add its identityto the end of the probe message so that when it returned to the initial sender, the complete cycle would be listed.

-The sender can then see which process has the highest number,

and kill that one or send it a message asking it to kill itself.

- Either way, if multiple processes discover the same cycle at the

same time, they will all choose the same victim.

Distributed Deadlock Prevention

- Consists of carefully designing the system so that deadlocks are structurally impossible
- Various techniques include:

  - allowing processes to hold only one resource at a time,

  - requiring processes to request all their resources initially, and

  - making processes release all resources when asking for a new one.

  -Most used: order all the resources and require processes to acquire

them in strictly increasing order.

  =This approach means that a process can never hold a high resource and ask for a low one, thus making cycles impossible.

- In a DS with global time and atomic transactions, two other practical algorithms are possible

  - based on the idea of assigning each transaction a global timestamp at the moment it starts.

  -essential that no two trans-actions are ever assigned exactly the same timestamp

Algorithms idea

- First idea:

  - when one process is about to block waiting for a resource that another process is using, a check is made to see which has a larger timestamp (i.e. is younger).

  - We can then allow the wait only if the waiting process has a lower timestamp (is older) than the process waited for.

  - In this manner, following any chain of waiting processes, the timestamps always increase, so cycles are impossible.

- Alternatively:

  - allow processes to wait only if the waiting process has a higher timestamp (is younger) than the process waited for, in which case the timestamps decrease along the chain.

- It is wiser to give priority to older processes.

  - They have run longer, so the system has a larger investment in them, and they are likely to hold more resources.

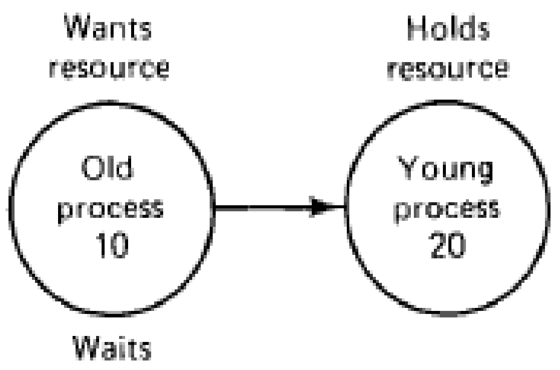
  - A young process that is killed off will eventually age until it is the oldest one in the system, so this choice eliminates starvation.

- Killing a transaction is relatively harmless, since by definition it can be restarted safely later.

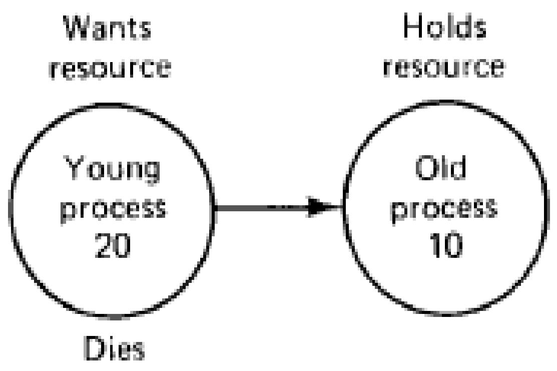
Example for the alg. wait-die

- (a): an old process wants a resource held by a young process.
- (b): a young process wants a resource held by an old process.
- In one case we should allow the process to wait; in the other we should kill it.
- Suppose that we label (a) dies and (b) wait.

  - Then we are killing off an old process trying to use a resource held by a young process, which is inefficient.
- Thus we must label it the other way, as shown in the figure.
- Under these conditions, the arrows always point in the direction of increasing transaction numbers, making cycles impossible.
- This algorithm is called *wait-die.*


Wants resource
Holds resource
Old process 10
Young process 20
Waits

(a)


Wants resource
Holds resource
Young process 20
Old process 10
Dies

(b)

Remark

- Assuming the existence of transactions,

  - it is possible to take resources away from running processes.
- When a conflict arises, instead of killing the process making the request, we can kill the resource owner.

  - Without transactions, killing a process might have severe

consequences, since the process might have modified files,

for example.

  - With transactions, these effects will vanish magically when the transaction dies.

Example for the alg. wound-wait

- Allow preemption
- A young whippersnapper will not preempt a venerable old sage
- One transaction is supposedly wounded (it is actually killed) and the other waits.
- If an old process wants a resource held by a young one, the old process preempts the young one, whose transaction is then killed
- The young one probably starts up again immediately, and tries to acquire the resource, forcing it to wait.
- Contrast with wait-die, where:

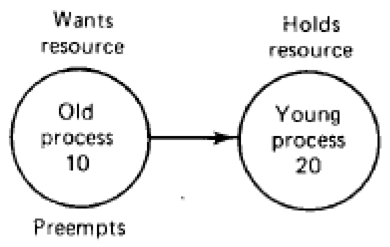
  -if an oldtimer wants a resource held by a young squirt, the oldtimer waits politely.

  - if the young one wants a resource held by the old one, the young one is killed.

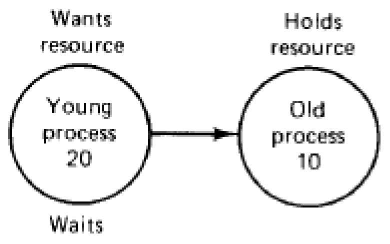
  - It will undoubtedly start up again and be killed again.

  - This cycle may go on many times before the old one releases the resource.

  Wound-wait does not have this nasty property.


Wants resource
Holds resource
Old process 10
Young process 20
Preempts

(a)


Wants resource
Holds resource
Young process 20
Old process 10
Waits

(b)

Application

- Used in Distributed Transaction and Concurrency Control

Scope of Research

- Novel Deadlock Detection Algorithm
- Deadlock avoidance for Distributed Real-Time and Embedded System