Deadlock

Multiple resources introduce the danger of deadlock.

A circle of processes each holding a resource wanted by another process in the circle.

It is a local phenomena
but it can easily spread.

Can it be cured?

Not without hurting some process.

At least one process must be forced to give up a resource it currently owns

(or provide a resource e.g. a message, which another process requires).

Conditions for deadlock

Havender's conditions for deadlock(1968) – 7.2.1

- There is a circular list of processes each wanting a resource owned by another in the list.
- Resources cannot be shared.
- Only the owner can release the resource
- A process can hold a resource while requesting another.

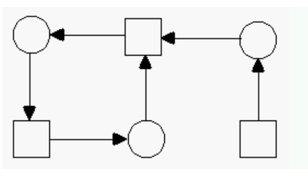
One reason deadlock is tricky is that testing may not discover it. It depends on the order of requests and allocations.

Deadlock detection

Resource graphs – 7.2.2

All resources and processes are vertices in the graph.

Allocations and requests are edges.



circles are processes, squares are resources

Cycles in the graph indicate deadlock (if each square holds one copy of the resource). It gets more complicated with multiple resources of the same type.
When should the deadlock detection algorithms run?

- How often do we expect deadlock?
- How many processes are usually affected?

Someone has to suffer

What do we do when deadlock is detected?

Remove a process

One of the processes in the circle can be selected and removed. Its resources are returned and the deadlock is broken.

We could use priority or age to select the process.

It may not solve the problem (deadlock may occur again immediately)

Remove all processes involved

Overkill but certainly solves the problem

Force a process to restart (or rollback to some safe state)

If we want to rollback, the system needs to maintain checkpoints where the processes can be restarted from.

We must ensure that the same process is not selected for restarting repeatedly.

Deadlock Prevention

We make sure at least one of the conditions will not be met. i.e. It is impossible for deadlock to occur if we use prevention.

There is a circular list of processes each wanting a resource owned by another in the list.

All resources must be issued in a specific order
if you have one of these you can't go back and request one of those.

an alternative

allow requests from earlier in the ordering if all resources later than this are returned first.

Resources cannot be shared.

Make them sharable?
Virtual devices - printer spooling

Deadlock prevention

Only the owner can release the resource

Forcibly remove - but that causes damage

or

if a new resource is currently busy release all currently held resources and try to get them back with the new one as well (the earlier resources may hence be removed)

or

if removing the resource temporarily does no harm, e.g. a page of memory or use of the CPU (state is saved and restored)

A process can hold a resource while requesting another.

Only allow one resource at a time?

or

Return a group of resources before requesting another group

or

Allocate all resources at once
can't ask for more as the process runs

Deadlock avoidance

Before granting requests we check if deadlock could occur if we allocate this resource to this process. “I can see deadlock might happen if I allow that. So I won’t allow that.”

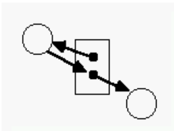
This may stop a process from getting a resource even though the resource is available.
But doing so leads to a situation which could cause deadlock later.
So avoidance prevents deadlock too - but dynamically as the processes run.

System knows who has what.
But doesn't usually know who wants what - has to use a very conservative strategy.
Worst case scenarios of future resource requests.

See section 7.5.2 for a simple algorithm (one of each type of resource).

Deadlock avoidance (cont.)

e.g. Two processes P and Q and two units of the same resource R.
Trouble only develops if both P and Q both require 2 Rs.



Obviously no problem if they each only want one R.
If P wants two Rs and Q has one (and doesn't want anymore), then P has to wait until Q releases its R.

Deadlock only occurs when they both have one and both want one more.

In this case the avoidance algorithm should not allow an allocation to the other process when one process already has an R.

The Banker's algorithm

Dijkstra invented a deadlock avoidance algorithm known as the Banker's algorithm.

- suppose that the request has been granted
- repeat until no more processes can be finished
  - search for a process which can be given all its resources
  - return all that process's resources to the system
- If all the processes can be removed then the state is safe and the allocation can go ahead.

Banker's example

e.g.
Two processes and two types of resource.
Two units of each resource.

	Resource A	Resource B
P's maximum demand	1	2
Q's maximum demand	2	1
Initial state	P0, Q0	P0, Q0
P requests A	P1, Q0	P0, Q0
P requests B	P1, Q0	P1, Q0
Q requests B		
Q requests A	P1, Q1	P1, Q0
P releases A	P0,Q1	P1, Q0
Q requests B	P0,Q1	P1, Q1

Disadvantages
We don't usually know the maximum resource requirements of each process.
Even if we do, this algorithm has to be performed every time a process requests a resource.

Distributed deadlock

As is usual, everything gets just a bit more complicated when dealing with distributed systems.

We can still prevent deadlock by resource ordering.

Or we can prevent deadlock by process ordering, only allowing processes with higher priorities to wait for resources. Processes with lower priorities get rolled back.

- But this quickly leads to starvation.

Or we can avoid deadlock by the Banker's algorithm, use one process as the banker:

- Even more expensive.

More processes, more resources, all requests have to be checked by a Banker process.

Time-stamp prevention methods

We prevent a cycle by only allowing older processes to wait for resources held by younger ones or vice versa. Rather than resource ordering this is process ordering.

wait-die

Process A requests a resource held by process B
If process A is older than process B it waits for the resource.
Otherwise process A restarts, process B (the older) continues.

Older processes hang around in the system (they have done more work). Age has its privileges.

Younger processes may have to restart multiple times, the resource might still be busy (but they eventually get old too, they retain their original timestamps).

Another time-stamp method

wound-wait

Once again A wants a resource held by B.

If A is older than B it takes the resource and B restarts.
Otherwise A (the younger) waits for B to release the resource.

Old processes never wait for anything. Age really has its privileges.

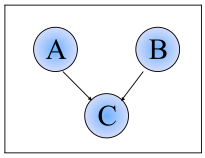
Less restarts.

In both cases processes keep their timestamps even when restarted.
Eventually they are really old and will not have to restart.

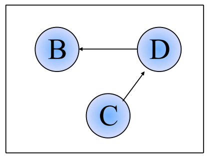
Either way lots of unnecessary restarts.

Distributed deadlock detection

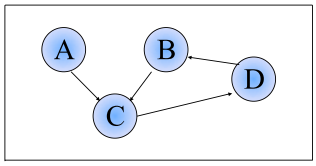
Each processor keeps track of the resource allocation graph to do with its local resources.
May include remote processes.
Cycles don't just occur locally.
Need to check the union of resource allocation graphs.



Site 1



Site 2



Global deadlock

DDD (cont.)

Centralized deadlock detection process.

Information may have changed by the time the data from the last machine is gathered, the data from the first machine is probably out of date.

Graph is only an approximation of the real allocation of resources and requests.
If there is deadlock it will be detected, but it is possible to detect deadlock when it doesn't exist.

Timestamps can be used to avoid false deadlock detection.

Avoiding false cycle detection

When process *A* at site *1*, requests a resource from process *B*, at site *2*, a request message with a timestamp is sent.

The edge $A \rightarrow B$ with the timestamp is inserted in the local graph of *1*. The edge with the timestamp is inserted in the graph of *2* only if *2* has received the request message and cannot immediately grant the requested resource.

The deadlock detection controller asks all sites for their wait-for graphs.

For requests between sites, the edge is inserted in the global graph if and only if it appears in more than one local graph (with the same timestamp).

Distributed approach

There is an extra node (P_{ex}) in each local wait-for graph.

All local processes waiting on any external processes point to P_{ex}.

Any local processes waited on by an external process are pointed to by P_{ex}.

If a cycle with P_{ex} involved is found we have a possible deadlock and information is sent to the site waited on.

If a deadlock is then found then it is handled.

If another possible deadlock is found involving P_{ex} another message is sent to another site etc.

Until either deadlock is detected or there is no cycle.

Before next time

Read from the textbook

11.1 File Concept

11.2 Access Methods

If you are interested in the internals of NTFS try

http://technet.microsoft.com/en-us/library/cc976808.aspx