Indefinite postponement and Deadlocks

Indefinite postponement

Conditions for deadlock

Solutions to the deadlock problem

A list of types of resources processes can be waiting for:

Reusable

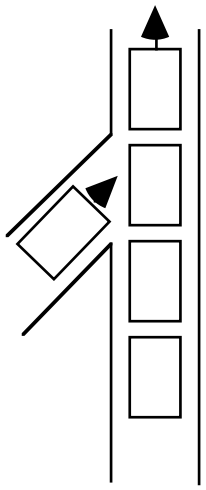
- memory
- processor
- devices
- semaphores
- files

Consumable

- message
- interrupt
- streams (e.g. input from a user)

Indefinite postponement waiting for a single resource

Lack of scheduling algorithms can cause indefinite postponement



bad luck, a process always just misses out

solution - employ a fair scheduling algorithm

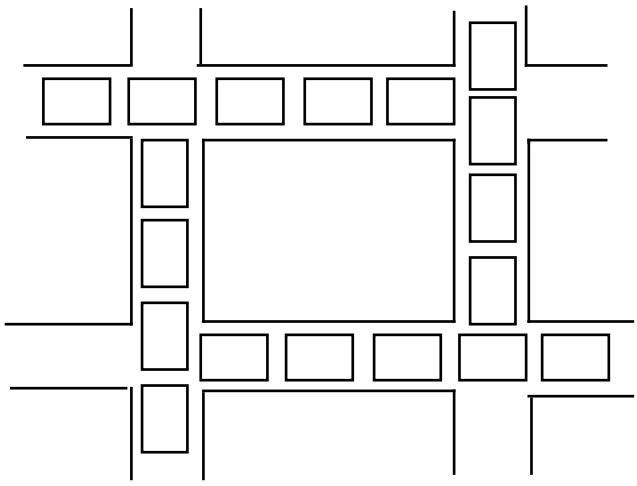
Scheduling algorithms can cause indefinite postponement

solution - make long waiting processes more likely to succeed

Multiple resources introduce the danger of deadlock

As when each philosopher held one fork.

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).

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.

Do as little damage as possible

If we have found deadlock in our system (which is not trivial)

Even though one request causes the deadlock to come about

all processes in the circle are guilty.

But one request might cause deadlock in several different circles.

In this case pick on this one.

Pick on less important processes to lose resources

(and probably all the work done so far)

Some characteristics to help us decide:

priority

time running

time remaining

number of resources

number of resources still required

the number of times this process has been restarted due to deadlock

Would like to restore the state from that process to what it was before hand

so that the process can restart and try again.

Detection

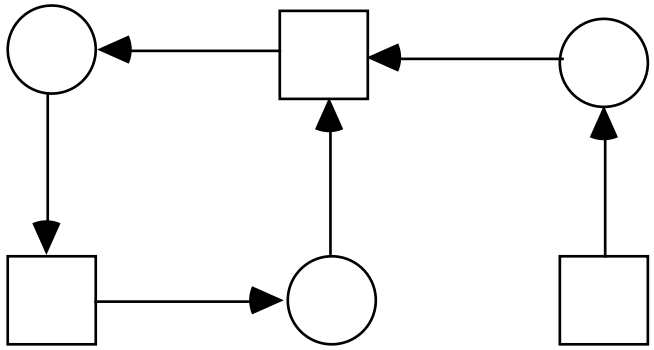
In order to know we have deadlock we need some method to detect it.

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.

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?

It may be better to stop deadlock occurring

Prevention

We need to guarantee that deadlock will never occur.

Each process can run without worrying - the system doesn't have to check.

Very expensive.

Need to make sure at least one of the conditions will not be met.

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 less than this are returned first

Resources cannot be shared.

Make them sharable?

Virtual devices - printer spooling

Only the owner can release the resource

Forcibly remove

Normally entails pain

or

to get a new resource release all currently held

and try to get them back with the new one as well

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)

<u>Avoidance</u>

Before granting requests we check to see if deadlock could occur

if we allocate this resource to this process.

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 request.

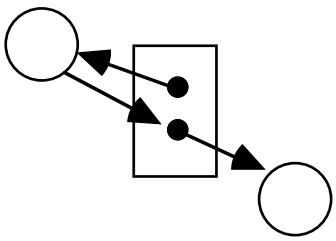
e.g.

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.