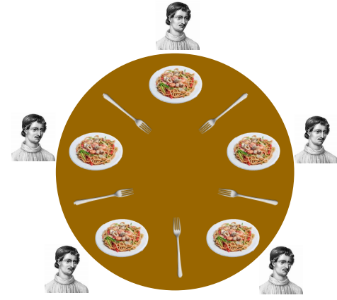
Operating Systems

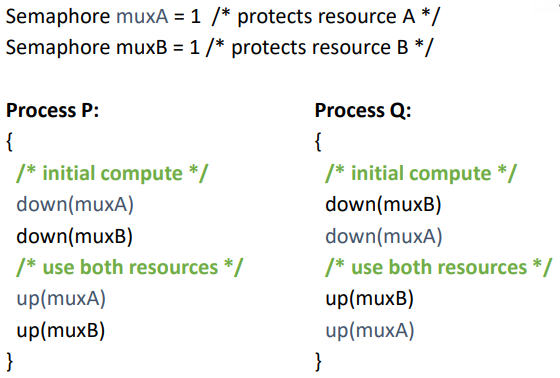
Deadlocks

The Dining Philosophers Problem

Imagine there are 5 philosophers sitting around a table with one chopstick between each philosopher (one chopstick per philosopher) and a philosopher must have two chopsticks in order to eat. Th philosophers alternate between thinking and eating.

Deadlock Problem

In the dining philosophers problem a deadlock will occur when all of the philosophers grab the chopstick to their left at the same time and all wait for the one at their right, this means they will all wait indefinitely.

Deadlock Example

Here we can see two processes that may get into a deadlock situation.

This will occur if both processes start at the same time:

* P – locks muxA
* Q – locks muxB
* P – waits on muxB
* Q – waits on muxA

And now neither will ever continue.

Deadlock

In a multiprogramming environment threads or processes compete for a finite number of resources each making requests for those resources. If resources are not available, the requesting thread/process will wait, the resources these processes are waiting on may be being held by other waiting threads/processes.

This is known as a type of liveness failure.

The definition of a deadlock is a situation in which every processs/thread in a set of processes/threads is waiting for an event that can only be caused by another process/thread in the set (and thus will never occur).

Livelock

A good analogue for livelock is to consider the situation where two people meet in a hallway, one moves to the right to allow the other through, but they move to the left, resulting in them remaining blocked, to solve this they then move to the other sides and stay blocked again (and so on).

Livelock is another kind of liveness failure similar to deadlock but the treads/processes are unable to proceed for different reasons.

A deadlock occurs when every thread/process in a set is blocked waiting for an even that can be caused only by another thread/process in the set, whereas livelock occurs when a thread/process continuously attempts an action that fails.

Necessary Conditions for Deadlock

1. Mutual exclusion
   * Each resource is either currently assigned to exactly one process/thread or is available
2. Hold and wait
   * Processes/threads currently holding resources that were granted earlier can request new resources
3. No pre-emption

* Resources previously granted cannot be forcibly taken away from a process/thread
  + - They must be explicitly released by the process/thread holding them

1. Circular wait
   * There must be a circular list of two or more processes/threads
     + Each waiting for a resource held by the next member of the chain

All of these conditions must hold for a deadlock to occur

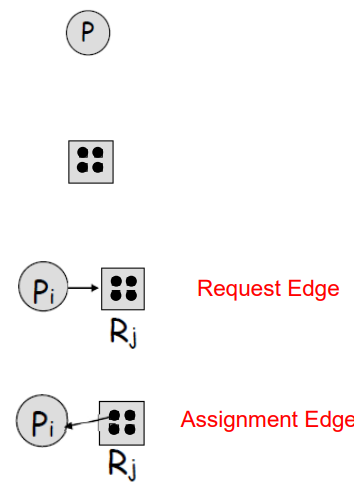
System Model (Resources and Processes)

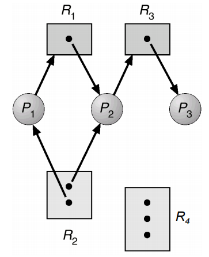
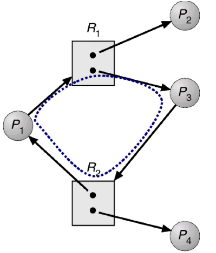
A system consist of a finite number of resources, partitioned into several types/classes, to be distributed among a number of competing threads/processes.

There can be any number (m) of types/classes of resources R1, R2, …, Rm with each resources type Ri having Ei instances. We assume these resources are reusable.

There can also be any number (n) of processes P1, P2, …, Pm instead of processes a model with threads can be developed.

Resource-Allocation Graph

Resource-allocation graphs are a way to visualise the current state of threads/processes (and resources).

RA graphs consist of a set of vertices V and a set of directed edges E. V is partitioned into two types, Process vertices (P, the set of processes) represented as circles, and Resource vertices (R, the set of resources) represented as rectangles (note these are just the types). E is also partitioned into two types, request edges directed from a process to a resources (process wants a resources of that type and may be blocked), and assignment edges directed from a resource to a process (process owns a resources of that type). Note that a resource vertex on its own is assumed to only have one instance, a if the resources vertex has dots in it, those dots represent its instances. Additionally request edges never go to instances whereas assignement edges always do.

The presence of cycles in these graphs can indicate the absence of deadlocks, if there are no cycles in the graph, then no thread/process is deadlocked. However if the graph does contain a cycle then a deadlock **MAY** exist, a cycle is not sufficient to imply a deadlock.

Note that:

* If each resource type has exactly oneinstance then a cycle does imply a deadlock has occurred.
* If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred.
* IF each resource type has several instances then a cycle doesn’t imply a deadlock has occurred, and is necessary but not a sufficient condition for a deadlock.

Handling Deadlocks

We can either ensure a deadlock will never occur or allow a deadlock to happen and deal with it then.

To ensure a deadlock never occurs we can either use prevention (done during development) to negate one of the four necessary conditions (usually the last) or avoidance (at runtime) where each resource request is analysed and denied if it may lead to a deadlock.

If we allow deadlocks to happen we can then either do nothing (most operating systems use this approach including Linux and Windows) or detect and recover from it (databases use this option).

Deadlock Prevention

To prevent deadlocks, we need to ensure that at least on of the necessary conditions cannot hold meaning deadlocks will be structurally impossible. This requires the way the software is written to be done carefully.

Attack the mutual exclusion condition:

The idea is to avoid assigning a resource unless absolutely necessary and to try to make sure that as few processes as possible may actually claim the resource. In practice, for read we don’t assign the resources to a single process (make it data read only) and for write we use a mediator that serialises the write (a spooler for example). The problem with this though is that processes may deadlock filling up the mediator.

Attacking the Hold and Wait condition:

The idea is to prevent the processes that hold the resources from waiting for more resources. In practice this requires all the processes to request all their resources before starting exertion, if everything is available, the process will be allocated whatever it needs and can run to completion, if one or more resources are busy, nothing will be allocated ant eh process will just wait (and then reallocate everything again. The problems with this are that processes don’t know all their resources and a process with many requirements may end up waiting indefinitely for them all to become available.

Attacking the No Pre-emption condition

The idea is that if a process is holding some resources and requests another resource that is not available, then all the resources it’s currently holding are pre-empted (released). In practice thi sis applied to resources whose state can be easily saved and restored later (CPU registers, memory space, etc…). The problem is that not all resources can be virtualised like this.

Attacking the Circular Wait condition

The idea is that all requests for resources must be made in numerical order by processes. In practice we give an ordering to the resources and can only requests a resource if the last resource we requested was lower than or equal to the desired resources in the ordering. The problem is that it may not be possible to find an ordering to satisfy everyone. Using this method our deadlock example from the beginning would have been impossible.

Deadlock Avoidance

Deadlock avoidance is achieved during the OS runtime and needs to be given additional information in advance of processes running such as what resources a thread/process will request and user as well as a complete sequence of requests and releases of each thread/process.

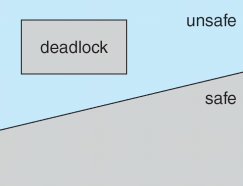
The system will then decided if a thread/process will wait for each request considering the resource-allocation state. The resource allocation state consists of the currently available resources, the currently allocated resources for each thread/process and future requests and releases of each thread/process.

The algorithm then dynamically examines the resource-allocation state for circular-wait conditions that may occur.

Safe State

If the system can allocate resources to each thread/process (up to its maximum) in some order and still avoid deadlocks then it that order is known as a safe sequence. The formal definition is that for a sequence <P1, P2, …, Pn> of all the processes in the systems, for each Pi, the resource that Pi can still request can be satisfied by currently available resources plus resources held by all the Pj with j < i. In this situation, if Pi’s resources are not immediately available, Pi can wait until all Pj have finished then Pi can obtain needed resources, execute, return allocated resources and terminate then when Pi terminates, Pi+1 can obtain its needed resources and so on.

If any safe sequence exists, then the system is said to be in a safe sate.

Safe, Unsafe, Deadlock States

A safe state can never be a deadlocked state, a deadlocked state is always an unsafe set, but an unsafe state isn’t necessarily a deadlock state, an unsafe state may lead to a deadlock.

In a safe state the os/runtime can avoid unsafe state (and therefore deadlocked) states. In an unsafe state the os/runtime cannot prevent threads from requesting resources in such away that a deadlock occurs.

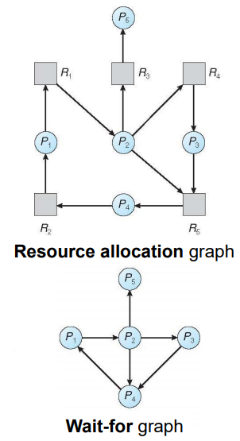
Threre are two types of deadlock avoidance algorithms, one that considers resource types that only have one instance (a variation of the graph algorithm) and one that considers resource types that can have multiple instance (the banker algorithm).

Banker’s Algorithm

This is an algorithm that could be used in a banking system to ensure the bank never allocates its available cash in such a way that it could no longer satisfy the needs of all its customers.

There is a set of controlled resources known to the system, the number of units of each resource is also known to the system and each application declares the maximum possible requirement of each resource type. When an application requests a set of resources the system determines whether the allocation of those resources will leave the system in a safe state, if it will then the resources are allocated, if not then you must wait until doing so will leave the system in a safe state.

Deadlock Detection and Recovery

We allow the system to enter a deadlock state, detection is done by an algorithm that examines the state of the system to determine whether a deadlock has occurred and then a recovery scheme is used (an algorithm to recover from the deadlock).

Detection in a system with a single instance of each resource type works as such:

We maintain a wait-for graph (a graph similar to the resource allocation graph, but where the resources are removed (and edges around them are collapsed)). We periodically search for cycles in the graph and if we find one we have a deadlock, this search requires n2 operations (where n is the number of vertices in the graph).

This can’t be used where there are mutliple instances of each resource type though.

Recovery from Deadlock

One solution is process/thread termination, the issue then is what do we terminate, do we abort all deadlocked process or just one process at a tie until the cycle is eliminated and if so how do we choose? This might leave the resources in an incorrect state though!

Another solution is resource pre-emption, we successively pre-empt resources from threads/processes and give them to others until the deadlock cycle is broken. This has a similar issue, how do we select a victim, there’s also the issues of how to roll them back and possible starvation where the same process/thread is the victim every time.

Instead of recovering we could always jut inform an operator and let them deal with it.