159.341 Programming Languages, Algorithms & Concurrency

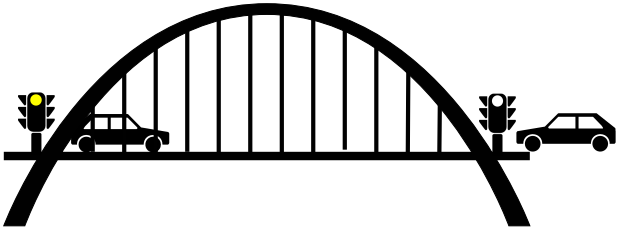
Deadlock (Part 1)

Daniel Playne
d.p.playne@massey.ac.nz

Deadlock

Reminder:
Some of the solutions proposed for synchronisation problems suffered from a risk of deadlock.

Critical-Section Problem - Deadlock



Left car:

```
while(true) {
   light[LEFT] = ON;
   while(light[RIGHT] == ON) {
      // wait
   }
   // cross
   light[LEFT] = OFF;
   // other tasks
}
```

Right car:

```
while(true) {
   light[RIGHT] = ON;
   while(light[LEFT] == ON) {
      // wait
   }
   // cross
   light[RIGHT] = OFF;
   // other tasks
}
```

Critical Section Problem - Deadlock

This approach solves the mutual exclusion problem - a car cannot enter the bridge at the same time as the other.

However, it does introduce another problem:

1. Left car arrives and turns its light ON
2. Right car arrives and turns its light ON
3. Left car waits for right light to turn OFF
4. Right car waits for left light to turn OFF
5. Both cars wait forever!

Deadlock

Deadlock is not just something that occurs when trying to build synchronisation primitives.

Even when using properly implemented mutexes and/or semaphores, deadlock can still occur.

Example - Dining Philosophers

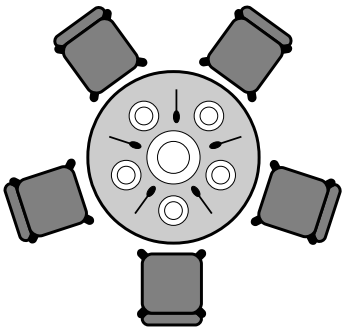
Dining Philosophers - Deadlock

The dining philosophers is a classic synchronisation problem.

Five philosophers sit around a table thinking and eating.

They can only eat while holding both the forks on their left and right.

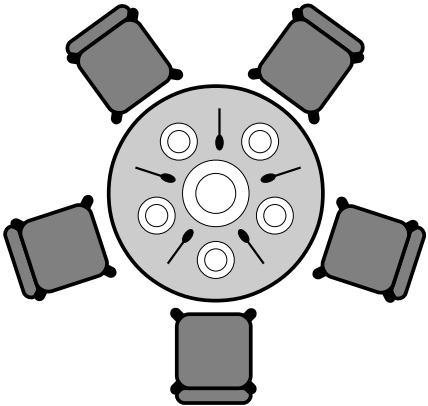
After they finish eating they put the forks back down.



Dining Philosophers.

Dining Philosophers - Deadlock

```
mutex m[5];

int left = id;
int right = (id + 1) % 5;
while(true) {
   // Think for a while
   think();

   lock(m[left]);
   lock(m[right]);

   // Eat using the spoons
   eat(spoon[left], spoon[right]);

   unlock(m[left]);
   unlock(m[right]);
}
```



Dining Philosophers.

Dining Philosophers - Deadlock

What happens if all the philosophers decide to eat at the same time?

They all pick up the spoon to their left, then wait to pick up the spoon to their right.

The philosophers get stuck and starve - **deadlock!**

Deadlock

Deadlock has occurred even though properly implemented mutex locks are being used.

Deadlock is obviously a serious problem for a concurrent programs as the entire program will be stuck indefinitely.

Luckily there are some conditions that we can identify that are necessary for deadlock to occur.

Deadlock - Model

The system we will consider consists of a finite number of resources that are to be distributed among a number of competing processes.

These resources may be classified into different types, when a process requests a resource of a particular type then **any** instance of that type can be used by the process to perform the necessary task.

If this isn't the case then the resources are incorrectly classified and should be different types.

Deadlock - Model

The synchronisation tools we have looked at (mutex locks and semaphores) can be considered a resource.

These are usually used to control access to a particular shared item so are normally considered to be separate types.

A process must obtain the appropriate lock protecting a shared object before accessing/modifying it.

Deadlock - Model

Processes must request a resource before it may start using it. After it has finished using the resource it must release the resource again. This normal order of operations for accessing a resource is:

1. **Request** - process requests a resource, if it cannot be immediately granted then the process must wait until is can acquire it.
2. **Use** - process uses the resource (whatever it is)
3. **Release** - process releases the resource.

Deadlock - Model

The different kinds of resources may have different ways of requesting, using and releasing them.

Access to files could be controlled through system calls `open()` and `close()`, or memory with `allocate()` and `free()`.

Our synchronisation tools had `lock()` and `unlock()` for mutex locks and `wait()` and `signal()` for semaphores.

Deadlock - Model

In deadlock, processes never finish executing and so the system resources they have requested will never be released which prevents other processes from being able to request those resources.

There are four necessary conditions for deadlock to occur in a system.

- **Mutual Exclusion** at least one resource must be held in a non-sharable mode (only one process at a time can use the resource). If another process requests the resource then it must wait until the resource is released.

Deadlock - Model

- **Hold and Wait** a process must be holding at least one resource and waiting to acquire additional resources currently held by other processes.
- **No preemption** processes cannot be preempted (resources are only released when the process holding it voluntarily releases it.
- **Circular Wait** a set of processes $\{P_0, P_1, ...P_n\}$ must exist such that each process P_i is waiting for P_{i+1} and P_n is waiting for P_0.

All four of these conditions must be true in order for deadlock to occur.

Resource Allocation Graph

A helpful way of representing deadlocks is with something called a system resource-allocation graph.

This is a directed graph representing the allocation of a set of resources R to a set of processes P.

The edges in the graph between these nodes represents the allocated and requested resources.

Resource Allocation Graph

A directed edge from a process P_i to a resource R_j represents that the process has requested access to an instance of the resource R_j and is currently waiting for it.

A directed edge from a resource R_i to a process P_j represents that an instance of that resource has been allocated to the process P_j.

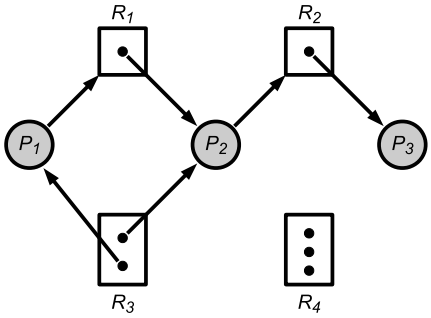
Resource Allocation Graph

As each resource may have multiple instances, each instance is represented by an additional dot inside the resource node.

A edge representing a request for an instance is made to the general resource node while an edge representing an allocation of the instance to a process is made from the instance dot.

Resource Allocation Graph

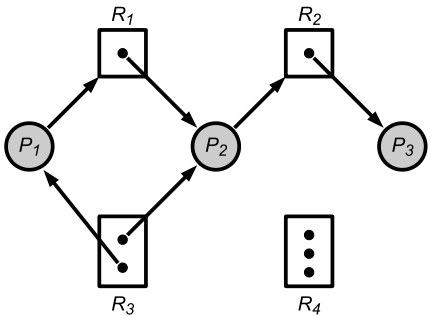
For example, a resource-allocation graph for resources $R = \{R_1, R_2, R_3, R_4\}$ and processes $P = \{P_1, P_2, P_3\}$.



R_1 and R_2 have one instance each, R_3 has two instances and R_4 has three instances.

Resource Allocation Graph

There are four instances of resources allocated:
$E = \{R_1 \to P_2, R_2 \to P_3, R_3 \to P_1, R_3 \to P_2\}$.



And two resource requests:
$E = \{P_1 \to R_1, P_2 \to R_2\}$.

Resource Allocation Graph

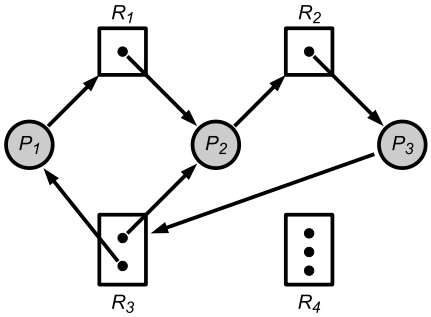
Given a resource-allocation graph, it can be shown that if the graph has no cycles then no process can be deadlocked.

If a cycle exists in the resource-allocation graph then a deadlock may exist.

If each resource type has only one instance then a cycle means that the system is deadlocked.

Resource Allocation Graph

This can be shown by adding an additional resource request to the resource-allocation graph.

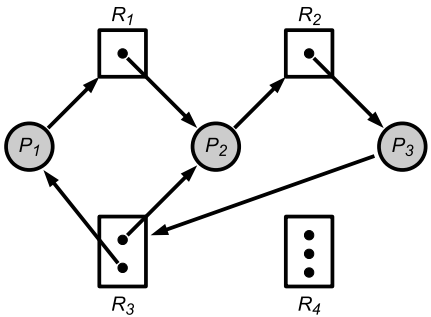


The process P_3 requests an instance of resource R_3.

Resource Allocation Graph

There are now two cycles in this resource allocation graph:

$P_2 \rightarrow R_2 \rightarrow P_3 \rightarrow R_3 \rightarrow P_2$
$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_2 \rightarrow P_3 \rightarrow R_3 \rightarrow P_1$



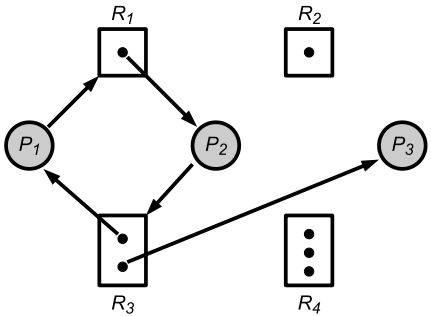
Resource Allocation Graph

At this point the system is deadlocked and no process can make any progress, each will need to wait until another process in the cycle releases a resource.

- P_1 cannot complete until it gets R_1 which is held by P_2.
- P_2 cannot complete until it gets R_2 which is held by P_3.
- P_3 cannot complete until it gets R_3 which is held by P_1 and P_2.

Resource Allocation Graph

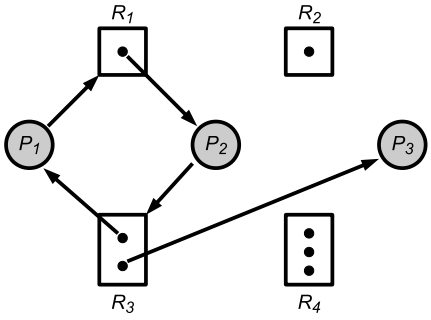
It should be noted that the presence of a cycle does not necessarily mean the system is deadlocked.



This resource-allocation graph has a cycle ($P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_1$) yet is not deadlocked.

Resource Allocation Graph

Process P_3 could release resource R_3. This will would P_2 to obtain R_3 which would allow it to complete and release both R_1 and R_3. Then P_1 will be able to obtain R_1 and also complete.



Resource Allocation Graph

Summary:

In general, if there is a cycle in a resource-allocation graph then the system **may** be in a deadlocked state.

If each resource has only **one** instance then the presence of a cycle means the system is in a deadlocked state.

If there is no cycle in the resource-allocation graph then the system is not deadlocked.

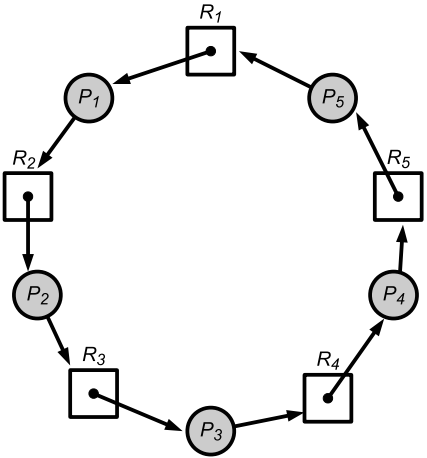
Resource Allocation Graph

Example: Draw a resource allocation graph showing a state of deadlock for the following code for the dining philosophers.

```
mutex m[5];

int left = id;
int right = (id + 1) % 5;
while(true) {
    // Think for a while
    think();

    lock(m[left]);
    lock(m[right]);

    // Eat using the spoons
    eat(spoon[left], spoon[right]);

    unlock(m[left]);
    unlock(m[right]);
}
```

Resource Allocation Graph

Each process is allocated the spoon to the left and is waiting for the spoon to the right.



Handling Deadlock

These definitions give some necessary conditions for deadlock to occur, but what to do about them?

There are three general approaches for dealing with deadlock:

- **Prevention/Avoidance** - ensure that the system never enters a deadlocked state.
- **Detection-Recover** - allow the system to enter a deadlocked state but then detect it and recover.
- **Ignore** - ignore the problem and hope it never happens.

Deadlock Prevention

The main idea behind deadlock prevention is to ensure that at least one of the conditions necessary for deadlock to occur cannot hold.

If we can ensure that just one of these conditions cannot hold then we can be sure that the system can never enter deadlock.

Deadlock Prevention

Mutual Exclusion - in some cases sharable resources do not require mutual exclusion (e.g. accessing read-only files) in which case deadlock will never occur.

However, this is not a general solution and many resources are fundamentally non-sharable. One of the requirements for the critical section problem was to provide mutual exclusion.

Deadlock Prevention

Hold and Wait - to ensure that the hold-and-wait condition never holds requires the system to guarantee that whenever a process holds a resource it does not also hold any other resource.

One policy for enforcing this would be to require each process to request and be allocated all resources before execution begins. This prevents a process from holding one resource while requesting another (it will either hold all required resources or none at all).

Deadlock Prevention

An alternative approach for preventing hold-and-wait from occurring would be to allow processes to request resources only if they currently hold none.

This allows a process to request resources during execution and then release them.

If a process holds resources and determines that it requires another, it must first release the resources it holds and then request all required resources at once.

Deadlock Prevention

No Preemption - another way to prevent deadlock would be to allow the system to preemptively take resources away from a process.

One protocol for this is that whenever a process requests a resource that cannot be immediately fulfilled, then all resources allocated to will be preemptively taken away.

The process can only continue when all requested resources are available.

Deadlock Prevention

These approaches can suffer from process starvation.

When a process requests two (or more) commonly requested resources it must wait until all the requested resources are available at once.

This condition may rarely (or never) occur and cause the process to starve.

Deadlock Prevention

Another way to implement this is as follows: whenever a process requests a resource that cannot be immediately fulfilled, the system checks to see if the process holding it is waiting. If so then the resource will be preemptively taken away from it and will be added to the list of resources that process is waiting for.

If the resource is neither available nor is the process holding it waiting then the process requesting the resource must wait.

Deadlock Prevention

These protocols can only be applied to resources whose state is able to be saved and restored.

They cannot be generally applied to resources such as mutex locks and semaphores as there will be no way of determining whether the shared state they are modifying is in a safe state or not.

Deadlock Prevention

Circular Wait - the final condition avoid that prevents deadlock. If we can ensure that there is no circular wait (a cycle in the resource allocation graph) then we can ensure that deadlock does not occur.

One protocol for doing this is assign a unique number to each of our different resources $\{R_1, R_2, ..., R_n\}$.

- $R_1 \rightarrow 5$
- $R_2 \rightarrow 12$
- $R_3 \rightarrow 15$
- $R_n \rightarrow 100$

Deadlock Prevention

When a process requests access to multiple resources, it must always do so in increasing order.

For example, if a process requires access to both resource R_1 and R_3 it must request them in that order because $R_1 \to 5$, $R_3 \to 15$ and $15 > 5$.

This can ensure that circular wait (cycles) can occur.

Deadlock Prevention

This scheme can be implemented at the application level by developers respecting this ordering when requesting resources.

There are some systems that analyse the order in which locks are obtained and generate warnings if processes access them in different orders.

Summary

- Deadlock Model
- Deadlock Conditions
- Resource Allocation Graph
- Deadlock Prevention

Chapters:

Silberschatz - chapter 7.1, 7.2 Tanenbaum - chapter 6.1, 6.2