Deadlock

Deadlock is only possible if certain *necessary* conditions are met: the *Coffman Conditions*

Deadlock

Deadlock is only possible if certain *necessary* conditions are met: the *Coffman Conditions*

1. **Mutual exclusion** Only one process can use a resource at a time

Deadlock

Deadlock is only possible if certain *necessary* conditions are met: the *Coffman Conditions*

1. **Mutual exclusion** Only one process can use a resource at a time
2. **Hold-and-wait** A process continues to hold a resource while waiting for other resources

Deadlock

Deadlock is only possible if certain *necessary* conditions are met: the *Coffman Conditions*

1. **Mutual exclusion** Only one process can use a resource at a time
2. **Hold-and-wait** A process continues to hold a resource while waiting for other resources
3. **No preemption** No resource can forcibly be removed from a process holding it

Deadlock

Deadlock is only possible if certain *necessary* conditions are met: the *Coffman Conditions*

1. **Mutual exclusion** Only one process can use a resource at a time
2. **Hold-and-wait** A process continues to hold a resource while waiting for other resources
3. **No preemption** No resource can forcibly be removed from a process holding it

All of these must hold for it to be *possible* to deadlock

Deadlock

A deadlock may be possible but will only actually happen if

Deadlock

A deadlock may be possible but will only actually happen if

4. **Circular Wait** There is a circular chain of processes where each holds a resource that is needed by the next in the circle

Deadlock

A deadlock may be possible but will only actually happen if

4. **Circular Wait** There is a circular chain of processes where each holds a resource that is needed by the next in the circle

This says that deadlock is happening as in the formal definition

Deadlock

It might seem easy to avoid these conditions, but in practice it's harder than you think

Deadlock

It might seem easy to avoid these conditions, but in practice it's harder than you think

Suppose we ensure Hold-and-wait never happens, e.g., requiring a process to drop other resources it holds whenever it gets blocked on a new request

Deadlock

It might seem easy to avoid these conditions, but in practice it's harder than you think

Suppose we ensure Hold-and-wait never happens, e.g., requiring a process to drop other resources it holds whenever it gets blocked on a new request

When it gets the new resource it will have to go back and pick up the other resources again

Deadlock

It might seem easy to avoid these conditions, but in practice it's harder than you think

Suppose we ensure Hold-and-wait never happens, e.g., requiring a process to drop other resources it holds whenever it gets blocked on a new request

When it gets the new resource it will have to go back and pick up the other resources again

Which may require it to drop the new resource while waiting. . .

Deadlock

It might seem easy to avoid these conditions, but in practice it's harder than you think

Suppose we ensure Hold-and-wait never happens, e.g., requiring a process to drop other resources it holds whenever it gets blocked on a new request

When it gets the new resource it will have to go back and pick up the other resources again

Which may require it to drop the new resource while waiting…

It is easy to get into a situation where the process never manages to get all the resources it needs: called *indefinite postponement*

Deadlock

Dining Philosophers

A popular illustration of deadlock is *The Dining Philosophers*

Deadlock

Dining Philosophers

A popular illustration of deadlock is *The Dining Philosophers*

Some Philosophers wish to share a plate of spaghetti, but they have only been provided with chopsticks

Deadlock

Dining Philosophers

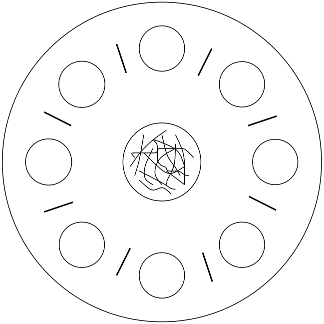
A popular illustration of deadlock is *The Dining Philosophers*

Some Philosophers wish to share a plate of spaghetti, but they have only been provided with chopsticks

Unfortunately, there is not quite enough chopsticks to go around

Deadlock

Dining Philosophers



Dining Philosophers

Deadlock

Dining Philosophers

Each Philosopher needs two chopsticks to eat, one from each side of their plate

Deadlock

Dining Philosophers

Each Philosopher needs two chopsticks to eat, one from each side of their plate

We have

Deadlock

Dining Philosophers

Each Philosopher needs two chopsticks to eat, one from each side of their plate

We have

1. Mutual exclusion. Only one Philosopher can use a chopstick at a time

Deadlock

Dining Philosophers

Each Philosopher needs two chopsticks to eat, one from each side of their plate

We have

1. Mutual exclusion. Only one Philosopher can use a chopstick at a time
2. Hold-and-wait. Each Philosopher wants to eat and won't let go of a chopstick until they have eaten

Deadlock

Dining Philosophers

Each Philosopher needs two chopsticks to eat, one from each side of their plate

We have

1. Mutual exclusion. Only one Philosopher can use a chopstick at a time
2. Hold-and-wait. Each Philosopher wants to eat and won't let go of a chopstick until they have eaten
3. No preemption. No-one is going to tell a Philosopher what to do!

Deadlock

Dining Philosophers

And if they all grab the left chopstick simultaneously

Deadlock

Dining Philosophers

And if they all grab the left chopstick simultaneously

4. Circular Wait. There is a circular chain of Philosophers where each holds a chopstick that is needed by the next in the circle

Deadlock

Dining Philosophers

And if they all grab the left chopstick simultaneously

4. Circular Wait. There is a circular chain of Philosophers where each holds a chopstick that is needed by the next in the circle

Of course, if the Philosophers were a bit more friendly, or polite, there would not be a problem

Deadlock

Dining Philosophers

And if they all grab the left chopstick simultaneously

4. Circular Wait. There is a circular chain of Philosophers where each holds a chopstick that is needed by the next in the circle

Of course, if the Philosophers were a bit more friendly, or polite, there would not be a problem

Exercise Identify the conditions in the car gridlock scenarios

Deadlock

There are two approaches to the problem of deadlock

Deadlock

There are two approaches to the problem of deadlock

1. Prevention. Stopping it happening ever by preventing one of the conditions occurring

Deadlock

There are two approaches to the problem of deadlock

1. Prevention. Stopping it happening ever by preventing one of the conditions occurring
2. Detection and Breaking. Letting deadlock happen, but spotting when it does and then breaking it by destroying one of the conditions

Deadlock

Prevention can be further refined

Deadlock

Prevention can be further refined

1a. Prevention. Constrain resource allocation to prevent at least one of the four conditions (e.g., ensure hold-and-wait never happens)

Deadlock

Prevention can be further refined

1a. Prevention. Constrain resource allocation to prevent at least one of the four conditions (e.g., ensure hold-and-wait never happens)
1b. Avoidance. Be careful not to allocate a resource if it can be determined that it might possibly lead to a deadlock in the future: keeping the system in a “safe” state

Deadlock

Prevention can be further refined

1a. Prevention. Constrain resource allocation to prevent at least one of the four conditions (e.g., ensure hold-and-wait never happens)

1b. Avoidance. Be careful not to allocate a resource if it can be determined that it might possibly lead to a deadlock in the future: keeping the system in a "safe" state

Avoidance is harder to manage as it needs to predict future requests for resources, but tends to be more efficient as it can allocate resources that prevention would disallow

Deadlock

Prevention

We can prevent deadlocks by disallowing any of the conditions

Breaking Mutual Exclusion

Deadlock

Prevention

Breaking Mutual Exclusion

This, quite often, cannot be broken

Deadlock

Prevention

Breaking Mutual Exclusion

This, quite often, cannot be broken

For example, trying to read a disk at the same time as another process is writing to it is a physical impossibility

Deadlock

Prevention

Breaking Mutual Exclusion

This, quite often, cannot be broken

For example, trying to read a disk at the same time as another process is writing to it is a physical impossibility

A lot of hardware only works if there is exclusive access, e.g., printers, sound cards, etc.

Deadlock

Prevention

Breaking Mutual Exclusion

This, quite often, cannot be broken

For example, trying to read a disk at the same time as another process is writing to it is a physical impossibility

A lot of hardware only works if there is exclusive access, e.g., printers, sound cards, etc.

Therefore we should take care to not hold on to such a resource for longer than is absolutely necessary

Breaking Hold-and-wait

Deadlock
Prevention

Breaking Hold-and-wait

We can require a process not to hold any resources if it ever gets blocked on another resource

Deadlock

Prevention

Breaking Hold-and-wait

We can require a process not to hold any resources if it ever gets blocked on another resource

This has the non-progress feature, as noted previously, and can be very inefficient with much grabbing and releasing to no avail

Deadlock

Prevention

Breaking Hold-and-wait

We might require a process to request all necessary resources simultaneously, blocking until all are available

Breaking Hold-and-wait

We might require a process to request all necessary resources simultaneously, blocking until all are available

- This might prevent the process from doing useful other work while one of the resources is unavailable but not yet needed by the process

Deadlock

Prevention

Breaking Hold-and-wait

We might require a process to request all necessary resources simultaneously, blocking until all are available

- This might prevent the process from doing useful other work while one of the resources is unavailable but not yet needed by the process
- Resources given to a process might be only needed much later, denying them to other processes in the meantime

Deadlock

Prevention

Breaking Hold-and-wait

We might require a process to request all necessary resources simultaneously, blocking until all are available

- This might prevent the process from doing useful other work while one of the resources is unavailable but not yet needed by the process
- Resources given to a process might be only needed much later, denying them to other processes in the meantime
- It may be that a process does not even know what resources it might need in advance, so this can be impossible to do anyway

Breaking Hold-and-wait

A variant of this was not even to admit a process to the scheduler until all resources are available: this is even worse

Deadlock

Prevention

Breaking Hold-and-wait

A variant of this was not even to admit a process to the scheduler until all resources are available: this is even worse

Perhaps a process only needs to write to disk at the end of a 2 hour compute session: do we really want to lock the disk for 2 hours?

Breaking No Preemption

Breaking No Preemption

This may only be possible for certain kinds of resource, namely those whose state can easily be saved and restored

Deadlock

Prevention

Breaking No Preemption

This may only be possible for certain kinds of resource, namely those whose state can easily be saved and restored

The OS might choose to preempt the holding process and take the resource away from it, giving it back later when the process is scheduled again

Deadlock

Prevention

Breaking No Preemption

This may only be possible for certain kinds of resource, namely those whose state can easily be saved and restored

The OS might choose to preempt the holding process and take the resource away from it, giving it back later when the process is scheduled again

This would be confusing for the holding process as the resource might change while it was owned by another process

Deadlock

Prevention

Thus, the resource should be given back to the process in an equivalent state to it was in when it was preempted, so the process can continue from where it left off

Deadlock

Prevention

Thus, the resource should be given back to the process in an equivalent state to it was in when it was preempted, so the process can continue from where it left off

For some resources this is possible, e.g., memory (see later)

Deadlock

Prevention

Thus, the resource should be given back to the process in an equivalent state to it was in when it was preempted, so the process can continue from where it left off

For some resources this is possible, e.g., memory (see later)

For others, not. For example, a printer

Breaking Circular Waits

Deadlock

Prevention

Breaking Circular Waits

One possible solution is to put an ordering on resources

$$R_1 < R_2 < R_3 < \dots$$

Deadlock

Prevention

Breaking Circular Waits

One possible solution is to put an ordering on resources

$$R_1 < R_2 < R_3 < \dots$$

E.g., (much simplified)

$$\text{disk 1} < \text{disk 2} < \text{printer} < \dots$$

Deadlock

Prevention

Then:

> *A process that holds resource R may then only request resources that are after R in the order*

Deadlock

Prevention

Then:

> *A process that holds resource R may then only request resources that are after R in the order*

In our example, if you have grabbed the printer, you cannot grab a disk

Deadlock

Prevention

Then:

> *A process that holds resource R may then only request resources that are after R in the order*

In our example, if you have grabbed the printer, you cannot grab a disk

If a process makes such a request, the OS simply refuses to grant it

Deadlock

Prevention

Then:

> *A process that holds resource R may then only request resources that are after R in the order*

In our example, if you have grabbed the printer, you cannot grab a disk

If a process makes such a request, the OS simply refuses to grant it

The process might choose to drop the printer and re-request the disk

Breaking Circular Waits

Now we cannot deadlock, as a deadlock would imply A has grabbed R_i and requested R_j; while B has grabbed R_j and requested R_i

Breaking Circular Waits

Now we cannot deadlock, as a deadlock would imply A has grabbed R_i and requested R_j; while B has grabbed R_j and requested R_i

For this to happen we would have both

$$i < j \text{ and } j < i$$

and this is impossible

Deadlock

Prevention

Breaking Circular Waits

This suffers the same problems as Hold-and-wait, namely inefficiency and unnecessarily holding resources

Deadlock

Prevention

Breaking Circular Waits

This suffers the same problems as Hold-and-wait, namely inefficiency and unnecessarily holding resources

Further, it works only if the process can make requests in increasing order; not always possible as it is not always possible to know what you need in advance

Breaking Circular Waits

This suffers the same problems as Hold-and-wait, namely inefficiency and unnecessarily holding resources

Further, it works only if the process can make requests in increasing order; not always possible as it is not always possible to know what you need in advance

And if you have R_1 and R_3, but then want R_2 you have to drop R_3, get R_2, then regain R_3; very inefficient

Breaking Circular Waits

This suffers the same problems as Hold-and-wait, namely inefficiency and unnecessarily holding resources

Further, it works only if the process can make requests in increasing order; not always possible as it is not always possible to know what you need in advance

And if you have R_1 and R_3, but then want R_2 you have to drop R_3, get R_2, then regain R_3; very inefficient

This usually effectively reduces to the request-all-at-once scenario

Deadlock

Avoidance

In contrast, deadlock *avoidance* does not break the conditions, but rather is careful not to do anything that might possibly create a deadlock in the future

Deadlock

Avoidance

In contrast, deadlock *avoidance* does not break the conditions, but rather is careful not to do anything that might possibly create a deadlock in the future

For each request, we have to decide whether granting the resource will potentially lead to a deadlock immediately or in the future

Deadlock

Avoidance

In contrast, deadlock *avoidance* does not break the conditions, but rather is careful not to do anything that might possibly create a deadlock in the future

For each request, we have to decide whether granting the resource will potentially lead to a deadlock immediately or in the future

Not so easy, as it requires knowing what might possibly happen in the future

Deadlock

Avoidance

In contrast, deadlock *avoidance* does not break the conditions, but rather is careful not to do anything that might possibly create a deadlock in the future

For each request, we have to decide whether granting the resource will potentially lead to a deadlock immediately or in the future

Not so easy, as it requires knowing what might possibly happen in the future

An unsafe request will not be granted by the OS

Deadlock

Avoidance

There are various algorithms that address the question of whether to grant a resource

Deadlock

Avoidance

There are various algorithms that address the question of whether to grant a resource

Exercise Dijkstra's Banker's Algorithm is one. Read about it and its limitations

Deadlock

Detection and Breaking

Next: deadlock *detection* systems allow deadlocks to happen but rely on noticing and breaking them

Deadlock

Detection and Breaking

Next: deadlock *detection* systems allow deadlocks to happen but rely on noticing and breaking them

The hope is that detection and breaking will be cheaper than avoidance: this is not always clear

Deadlock

Detection and Breaking

Next: deadlock *detection* systems allow deadlocks to happen but rely on noticing and breaking them

The hope is that detection and breaking will be cheaper than avoidance: this is not always clear

The earliest detection system was *Detection by Operator*

Deadlock

Detection and Breaking

Next: deadlock *detection* systems allow deadlocks to happen but rely on noticing and breaking them

The hope is that detection and breaking will be cheaper than avoidance: this is not always clear

The earliest detection system was *Detection by Operator*

“The machine seems to have stopped. . . ”

Deadlock

Detection and Breaking

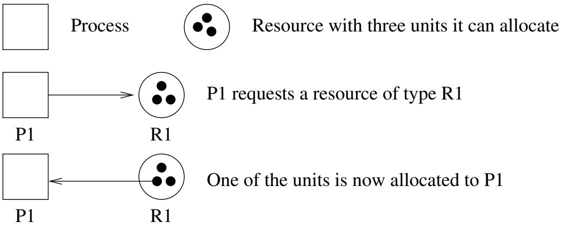
The chief method employed is to spot when the circular wait happens

Deadlock

Detection and Breaking

The chief method employed is to spot when the circular wait happens

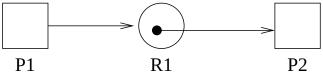
One method for deadlock detection uses *resource request and allocation graphs* (RRAG)



RRAGs

Deadlock

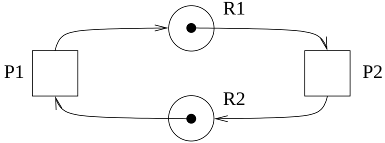
Detection and Breaking



P1 requests from R1, but it has no free units, so P1 will be blocked

Deadlock

Detection and Breaking



Circular Wait

P1 requests from R1, but it has been allocated to P2;
P2 requests from R2, but it has been allocated to R1:
this is deadlock

Deadlock

Detection and Breaking

So deadlock detection is just finding these kinds of loops in RRAGs

Deadlock

Detection and Breaking

So deadlock detection is just finding these kinds of loops in RRAGs

This can be done by a fairly simple *graph reduction* method that takes a graph and removes certain edges until either (a) there are no edges left or (b) a loop is found

Deadlock

Detection and Breaking

So deadlock detection is just finding these kinds of loops in RRAGs

This can be done by a fairly simple *graph reduction* method that takes a graph and removes certain edges until either (a) there are no edges left or (b) a loop is found

Exercise Read about this

Deadlock

Detection and Breaking

So that leaves breaking the deadlock: as always there are lots of ways we can do this, none terribly satisfactory

Deadlock

Detection and Breaking

So that leaves breaking the deadlock: as always there are lots of ways we can do this, none terribly satisfactory

- Kill one or more or all of the deadlocked processes: a bit drastic, but sometimes the only solution. But which process? For example, out of memory “OOM killers” are tricky to get right

Deadlock

Detection and Breaking

So that leaves breaking the deadlock: as always there are lots of ways we can do this, none terribly satisfactory

- Kill one or more or all of the deadlocked processes: a bit drastic, but sometimes the only solution. But which process? For example, out of memory “OOM killers” are tricky to get right
- Preempt the blocking resources: better, if possible. If there are multiple resources causing the deadlock we have to choose which, as preempting just a few might free things up enough

Deadlock

Detection and Breaking

So that leaves breaking the deadlock: as always there are lots of ways we can do this, none terribly satisfactory

- Kill one or more or all of the deadlocked processes: a bit drastic, but sometimes the only solution. But which process? For example, out of memory "OOM killers" are tricky to get right
- Preempt the blocking resources: better, if possible. If there are multiple resources causing the deadlock we have to choose which, as preempting just a few might free things up enough
- Add resources: rarely possible

Exercise Think about how you might apply deadlock prevention or breaking to (a) Dining Philosophers and (b) the car deadlock scenarios

Deadlock

In real life, a popular approach is simply to ignore the possibility of deadlock happening

Deadlock

In real life, a popular approach is simply to ignore the possibility of deadlock happening

Sometimes called the *Ostrich Algorithm*

Deadlock

In real life, a popular approach is simply to ignore the possibility of deadlock happening

Sometimes called the *Ostrich Algorithm*

There is not entirely stupid, as it argues that the costs associated with prevention or detection are large, and if deadlocks are rare, then the cost of an occasional reboot of the machine is small in comparison

Deadlock

In real life, a popular approach is simply to ignore the possibility of deadlock happening

Sometimes called the *Ostrich Algorithm*

There is not entirely stupid, as it argues that the costs associated with prevention or detection are large, and if deadlocks are rare, then the cost of an occasional reboot of the machine is small in comparison

In a carefully written OS, you can eliminate many of the possible causes of deadlock, or, at least, reduce the chances of them happening

Deadlock

Some resources are preemptable, e.g., memory (as we shall discuss in depth later), but a more general solution (which also applies to memory) is *virtualisation*, where the OS pretends each process has sole access to a resource

Deadlock

Some resources are preemptable, e.g., memory (as we shall discuss in depth later), but a more general solution (which also applies to memory) is *virtualisation*, where the OS pretends each process has sole access to a resource

We have already seen this for printers in the form of *spooling*

Deadlock

Some resources are preemptable, e.g., memory (as we shall discuss in depth later), but a more general solution (which also applies to memory) is *virtualisation*, where the OS pretends each process has sole access to a resource

We have already seen this for printers in the form of *spooling*

A process thinks it is writing to a printer, but it is actually writing to a tape, and the tape is later written to the printer

Deadlock

Similarly, for example, a process thinks it writes to a network card but the data is actually buffered by the OS somewhere in memory, to be sent later when the card is free

Deadlock

Similarly, for example, a process thinks it writes to a network card but the data is actually buffered by the OS somewhere in memory, to be sent later when the card is free

And so on for other kinds of devices: a process interfaces with its own virtualised device, there is no possibility of deadlock as every process can progress without waiting, and the OS sorts out transferring the data to or from the real device

Deadlock

Similarly, for example, a process thinks it writes to a network card but the data is actually buffered by the OS somewhere in memory, to be sent later when the card is free

And so on for other kinds of devices: a process interfaces with its own virtualised device, there is no possibility of deadlock as every process can progress without waiting, and the OS sorts out transferring the data to or from the real device

But, of course, this new perspective just shifts the actual problem: when and in what order should the OS do the I/O?

Deadlock

Similarly, for example, a process thinks it writes to a network card but the data is actually buffered by the OS somewhere in memory, to be sent later when the card is free

And so on for other kinds of devices: a process interfaces with its own virtualised device, there is no possibility of deadlock as every process can progress without waiting, and the OS sorts out transferring the data to or from the real device

But, of course, this new perspective just shifts the actual problem: when and in what order should the OS do the I/O?

This is called *I/O scheduling*

Deadlock

Similarly, for example, a process thinks it writes to a network card but the data is actually buffered by the OS somewhere in memory, to be sent later when the card is free

And so on for other kinds of devices: a process interfaces with its own virtualised device, there is no possibility of deadlock as every process can progress without waiting, and the OS sorts out transferring the data to or from the real device

But, of course, this new perspective just shifts the actual problem: when and in what order should the OS do the I/O?

This is called *I/O scheduling*

Exercise Virtualisation allows the OS to prevent deadlocks. So which of the Coffman Conditions does it disallow?