Deadlocks

Suppose we have two threads, 1 and 2. Each one needs to use two locks, and they run the code below.

Thread 1	Thread 2
`lock1.acquire()`	`lock2.acquire()`
`lock2.acquire()`	`lock1.acquire()`

Suppose that Thread 1 runs and acquires `lock1` and then a context switch happens to Thread 2. Thread 2 runs and acquires `lock2` and then has to wait for `lock1` because Thread 1 has it. Then we context switch back to Thread 1, which tries to acquire `lock2`. It can't get it because Thread 2 has it. And now we're stuck. Thread 1 has `lock1` and needs `lock2`, while Thread 2 has `lock2` and needs `lock1`. Each thread has a lock the other needs and both are stuck waiting for each other's locks. This is called *deadlock*. Neither thread can do anything. Here a few real-life examples of deadlocks.

1. Two people, A and B, are interested in each other, but they're both shy. Person A is waiting for B to talk to them and person B is waiting for person A to talk to them. Neither will make any progress because they are both waiting for each other.

2. Suppose a printer is configured so that it won't print a file until it has the entire thing. Two people, A and B, both send large files to the printer at the same time. Individually, either file is can fit in the printer's memory, but combined they are too large. Midway through the upload, we end up with half of A's file and half of B's file in memory, and memory is filled up. The system is locked up and no one can print anything.

3. Network routers sometimes end up receiving traffic faster than they can process it. They have a buffer to store incoming packets, and other routers won't send things to a router if they know its buffer is full. Suppose we have three routers, A, B, and C, where A sends things to B, B sends things to C, and C sends things to A, and suppose also that everyone's buffers are full. Then we're at a deadlock. Nobody can send anything to anyone else because everyone is full, and nobody can free up any space because they need to send stuff in order to do that. So the whole system is locked up and grinds to a halt.

In general, here is one way for a deadlock to happen with two individuals and two resources:

A has 1 and needs 2
B has 2 and needs 1

In this case, each process is preventing the other from moving forward, so they are both stuck. And here is a deadlock involving three individuals and three resources:

A has 1 and needs 2
B has 2 and needs 3
C has 3 and needs 1

Here B prevents A from making progress, C prevents B from making progress, and A prevents C from making progress. So, again, everyone is locked up.

How to deal with deadlocks

In this section, we'll look at some strategies for dealing with deadlocks. Deadlocks can happen in a variety of scenarios. Below we'll just talk about processes and resources, but what is here applies to many scenarios, like threads and locks, people and printers, etc.

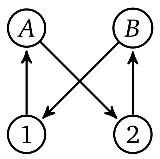
Ignore them Believe it or not, one way to deal with deadlocks is to ignore them. Just because a deadlock is possible, that does not mean it will happen. It might be the sort of thing that only happens once a year, and when it does, you just restart the system. Often, a deadlock will not occur unless a very unlucky sequence of events happens.

It might happen that trying to write code to ensure that a deadlock doesn't happen ends up introducing all sorts of other bugs and problems that are worse than the occasional deadlock. Or it might be that there are better things you can spend your time on than worrying about deadlocks. This isn't always a viable solution, but sometimes ignoring things really is the best option.

Detect and recover Here we will try to detect when a system is deadlocked and then figure out what to do about it. One way to detect deadlocks is to build a particular directed graph. The vertices of the graph are both processes and resources. Edges (arrows) are as follows:

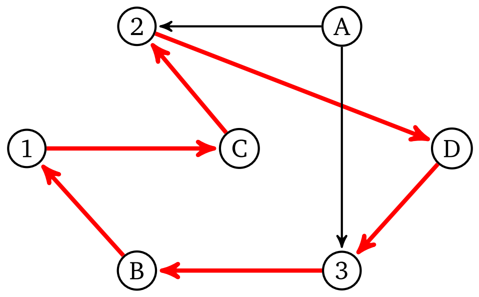
- If a process needs a resource, we add an edge from the process to the resource.
- If a resource is held by a process, we add an edge from the resource to the process.

A deadlock is indicated by a cycle in the graph, where we start at a vertex, follow directed edges from it, one after another, and end up eventually back at the start. For example, suppose the processes are called *A* and *B* and the resources are called 1 and 2. Let's say that *A* has 1 and needs 2, while *B* has 2 and needs 1. The graph below shows this:



We see that there is a cycle that goes from *A* to 2 to *B* to 1 and back to *A* again. This indicates a deadlock. In particular, *A* is stuck waiting for a resource that *B* has while *B* is waiting for a resource *A* has. Neither can do anything because they spend all their time waiting for a resource the other one has.

Here is a larger example:



This graph is a pictorial description of the following info:

- *A* needs 2 and 3
- *B* has 3 and needs 1
- *C* has 1 and needs 2
- *D* has 2 and needs 3

In the graph, the cycle is highlighted. It goes from *B* to 1 to *C* to 2 to *D* to 3 and back to *B* again. This indicates a deadlock. Note that there is no deadlock involving *A* because there aren't any paths we can take starting at *A* and following the arrows that will get us back to *A*.

Once we have detected a deadlock, to recover from it, there are a few possibilities:

1. Stop all processes and restart the system.
2. Maybe just stopping a single process will break the deadlock.
3. Maybe just removing a resource from a process will break the deadlock.
4. Checkpointing — we can roll the system back to a point at which there was no deadlock.

For all of these possibilities, in a real system, we have to be careful. For instance, restarting a nuclear reactor to stop a deadlock is not recommended. And when stopping a single process or removing a single resource, we have to be careful which one we choose, as certain processes are more important than others.

The four necessary conditions It's a provable mathematical fact that a deadlock can only happen if *every one* of the following conditions is met.

1. Mutual exclusion — Only one process can hold a resource at a time.
2. Hold and wait — Processes that currently have resources are allowed to request more.
3. No preemption — We can't forcibly remove a resource from a process.
4. Circular wait — There is a cycle, like in the previous section, where A is waiting for a resource that B has, B is waiting for a resource that C has, etc., all the way back to a last process that is waiting for a resource that A has.

These are called necessary conditions because they are needed in order for there be a deadlock. Since every one of these has to be simultaneously true, if we can somehow make any one of these false, then we can prevent deadlocks. Below are some ways to remove them.

1. Removing mutual exclusion — Obviously if resources are allowed to be shared, this isn't a problem. But if they can't be shared, then one approach is for the system to be stingy in handing out resources. The system could only give out resources to processes that absolutely need them or it could limit who is allowed to have a resource. A lot of real-life bosses behave this way.
2. Removing hold and wait — One thing is to require a process to acquire all the resources it needs right when it starts running. If it can't do that, then it has to wait until they are all available. One issue with this is that it's often hard to know what you'll need ahead of time, and this wouldn't work well if there are several long-running processes on the system because if they have similar resource needs, then you could only have one of them running at a time. But solutions like this are often used in database and transaction processing.

   Another approach is that whenever a process needs a new resource, it has to drop all the resources it currently has and then reacquire them. This sometimes works, but it of course means that another process might immediately grab what was dropped.
3. Removing no preemption — This is pretty simple. Just allow the system to take back resources. Sometimes this works, but it could make things unstable. For instance, if you take back a lock on a file while a process is writing to it, the file contents could be corrupted.
4. Removing circular wait — One option is to only allow processes to have one resource at a time. If they want a new resource, they have to drop the one they currently have. Of course, this won't work if processes need multiple resources at a time.

   Another option is to have a specific order in which requests for resources have to be made. For instance, here is the code from the beginning of this section that causes a deadlock:

Thread 1	Thread 2
`lock1.acquire()`	`lock2.acquire()`
`lock2.acquire()`	`lock1.acquire()`

If we require that lock1 must always be acquired before lock2, then the two threads would have to be rewritten like this:

Thread 1	Thread 2
`lock1.acquire()`	`lock1.acquire()`
`lock2.acquire()`	`lock2.acquire()`

Now there is no chance of a deadlock. If a context switch happens in between Thread 1 acquiring lock1 and lock2, there is no issue because Thread 2 will immediately stop when trying to acquire lock1. A context switch will back to Thread 1 will happen and it will be able to get lock2. If two threads running on separate CPUs both get to the `lock1.acquire()` line at the same time, because of how the test-and-set lock is designed, only one thread will get the lock and the other will have to wait. Here is another example with three threads:

Thread 1	Thread 2	Thread 3
`network.acquire()`	`printer.acquire()`	`printer.acquire()`
`database.acquire()`	`database.acquire()`	`network.acquire()`
`printer.acquire()`	`network.acquire()`	`database.acquire()`

We have locks for resources like a printer, an important file, and a shared network connection, where in this system only one thread is allowed to be using these at a time. And it might not be that each call to acquire runs immediately after the previous one. Maybe Thread 1 connects the network then needs to save the downloaded info to the network and some time after that needs to print something. Thread 2 maybe is doing a different task that requires the resources be accessed in a different order. Suppose Thread 1 acquires the network and database locks and a short while later Thread 2 acquires the printer lock. At this point, the system is deadlocked, or will be shortly. Threads 1 and 3 both need the printer lock and can't get it since Thread 2 has it. Thread 2 needs the database lock, but Thread 1 has it. We can fix this problem by requiring all threads follow the same order. For stopping deadlocks, it doesn't matter what that order is, as long as it's consistent from thread to thread. For example, here is one possible ordering.

Thread 1	Thread 2	Thread 3
`nework.acquire()`	`network.acquire()`	`network.acquire()`
`database.acquire()`	`database.acquire()`	`database.acquire()`
`printer.acquire()`	`printer.acquire()`	`printer.acquire()`

So, if threads need the network, database, and printer, they always have to acquire them in that order. You could try to go through and find a sequence of events that would cause a deadlock above, but you'll see it's not possible. In general, just make sure everyone acquires things in the exact same order and there is no threat of a deadlock. This is a nice solution, but sometimes it's hard to come up with an ordering that works well for everyone involved.

Livelock

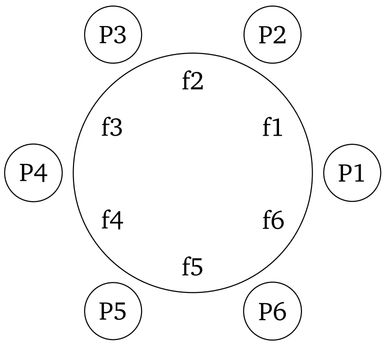
A deadlock is where multiple processes are completely stopped waiting for one another. A *livelock* is a similar situation where multiple processes can't make any forward process, but the difference is that they are still moving in a sense.

For example, a deadlock in a busy hallway might happen where things get so crowded that no one is able to move, and no one can even back up to get out of anyone else's way. A livelock would be where two people are walking down a hallway right toward each other. Person A starts moving to the left to get out of the way and at the same time Person B does that. So A then moves to the right and B does the same. You've probably been in a situation like this. If A and B keep trying to move out of each other's way and keep getting stuck like this, that's a livelock. Here the "live' part of things is because they are actually moving, while in a deadlock, there is no motion.

In a computer scenario, this could be where A has resource 1 and needs 2, and B has resource 2 and needs 1. In order to help B out, A decides to drop resource 1. At the same time, B decides to drop resource 2. Then each process sees that the resource they need is free, and so A picks up 2 and B picks up 1. Imagine repeating this process over and over, and we have a livelock.

The Dining Philosophers Problem

This is a famous problem about deadlocks, introduced by the computer scientist Edsger Dijkstra. There are several philosophers all sitting around a table. Between each philosopher there is a fork, like below:



Each philosopher can do only two things: eat and think. In order to eat, a philosopher needs to pick up both the fork to their left and the fork to their right. So each philosopher is basically a thread running the following code:

```
while True
    think()
    grab_forks()
    eat()
    drop_forks()
```

If we're not careful, we can get a deadlock. This can happen if they all decide to eat at once. If each one grabs their left fork first and then their right fork, we'll be in a situation where each philosopher has exactly one fork and therefore can't eat.

There are several solutions to this. One is to make it so that one of the philosophers is forced to grab their fork in a backwards right-left order. Another solution is to use locks. A weakness of this is only one philosopher will be able to eat at a time, even though there are enough forks for more than one to be eating at the same time.

There is a better solution that uses locks and semaphores. We create a boolean variable for each philosopher indicating whether or not they are currently eating. We also have a semaphore for each one, initially set to 0. And there is a lock to protect the critical section so that they when they are ready to eat, they will wait until the forks are available.

This might seem like a bit of a contrived problem, and it is, but it represents a simplified model of real-life situations. It's also easier to reason about something like this than trying to reason about a deadlock in 200 lines of messy C code.