A system is said to be in "deadlock" whenever it is stuck in a state where each part of the system is waiting for some event that can only be caused/triggered by another part of the system. (which is also waiting for another part, etc.) The various parts wait on each other, and no progress can be made (the system state cannot change).
Deadlock in computer systems

- Deadlocks can occur in various situations – the idea is not specific to operating systems or to IO.
- We will specifically be dealing with "resource deadlock". Gridlock (traffic) is an example of deadlock, but not an example of "resource deadlock".
- "Resource" is a very general term roughly meaning "something that will be used towards some goal".
- An I/O device (eg. printer) can be seen as a resource that some programs may use to do their job.
- A resource deadlock occurs eg. when process A of a system currently holds (exclusively controls) resource 1 and process B of the system holds resource 2, but A is currently waiting to use resource 2 while B is also waiting to use resource 1.
- The concept of deadlock is therefore relevant to chapter 3.
Resource preemption

Resources can generally be classified into two types. This is related more to the way the device is used, and is not necessarily in an intrinsic characteristic of that device.

- **A preemptable** resource is one that can be taken away from the process owning it with no ill effects.
- **A nonpreemptable** resource, is one that cannot be taken away from its current owner without causing the computation to fail.

For example: taking access to the CDRW drive away from a process that is currently busy burning data to a CD will probably *not* damage the computer, but it will result in a "corrupted" CD that cannot be used.

The CD writer can in this situation be seen as a non-preemptible resource. While the CD is burning, the program (eg. Nero) is "holding"/"locking" the CD writer resource.

Deadlock usually involves non-preemptible resources.
Formal definition of deadlock

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

Resource deadlock can potentially become a problem if and only if all of the following conditions are true for a given system:

1. **Mutual exclusion**: a resource can be held by only one process at a time.
2. **Hold and wait**: a process that holds one or more resources may request more/other resources as well.
3. **No preemption**: resources will never be taken away from a process. A resource must first be released by the process using it before another can use it.
4. **Circular wait**: it must be possible for a set of processes to get into a state where each process is waiting for another process to release a resource.
Modelling deadlock

Process and resource (IO device) acquisition can be depicted via directed graphs.

Such a representation can be used to understand deadlocks. Here, process A is already holding resource R, and process B is currently trying to acquire resource S.

No other process is controlling S, so B's request will be granted by the OS (unless something else went wrong eg. S malfunctions)

Of course, there occurs no deadlock in this situation
Here, process D is trying to acquire U, but it is already being used by C, while C is waiting for T, but it is being used by D.

Deadlock!

Clearly, there is a cycle in the graph. Condition #4 (circular wait) basically states that deadlock occurs (only) when there is a cycle in such a process/resource graph.

It may therefore seem simple to detect or prevent deadlock – just look at the current process/resource graph. But this can become intractibly complicated, even for small graphs.
Just because a deadlock CAN happen does not mean it WILL happen. It will depend on:

- the specific order in which processes request resources
- the order in which the system scheduler decides to execute those processes.

There is a good illustrated example on page 243 of the textbook (240 in pdf version): depending on the order in which processes are scheduled, the circular wait condition may or may not be fulfilled, and a deadlock may or may not occur.
Dealing with deadlocks

There are several ways to deal with the problem of deadlocks. Such methods can generally be classified as:

1. **Detection and recovery**: allow deadlocks to occur, but provide some way to detect and recover from them.

2. **Prevention**: remove one of the four conditions necessary for deadlocks to occur.

3. **Avoidance**: let the OS carefully manage the way resources are allocated, in order to prevent deadlocked states.

4. **Don't worry about it**: do not implement a specific mechanism in the OS to deal with deadlocks. Surprisingly, this is often the best solution especially when deadlocks are unlikely to happen.
Detection and recovery

• A simple (but imprecise!) way to detect deadlocks is simply to monitor how long processes wait for resources. If a process is waiting for longer than a certain time (e.g., 10 min) then the OS can conclude that that process is probably part of a deadlock => terminate that process.

Of course, some processes that wait for or hold resources for a long time because of long computations might be "diagnosed" incorrectly and stopped unnecessarily.

• A better way would be for the OS to construct a graph of processes and resources, and monitor the graph for cycles. The OS then terminates deadlocked processes to break cycles. Often impractical because it would require extra computational overhead in the OS, slowing down the system.
Deadlock prevention

Remove (negate) one of the conditions necessary for deadlock

#1 Mutual exclusion: if all resources on the system are made fully shareable, no deadlock can occur. Practical for some simple computer systems, but can't/shouldn't be done on systems that are better off with some form of resource locking.

#2 "Hold and wait" can be attacked in different ways:

1) Require processes to request ALL their resources at once when starting up. Any process will either fail to start up OR acquire its resources, run to completion, and release its resources for the next process.

   **Problems:** inefficient, not all process can know beforehand how many resources they may require.
Deadlock prevention

#2 "Hold and wait" can be attacked in different ways:

2) A process may only acquire new resources after releasing its current resources.
   This is a more practical approach.

Problems:
Not applicable in some cases,

3) Can potentially lead to "resource starvation" where one process keeps waiting for a very long time before enough resources are freed up for it to use.

#3 "No preemption" cannot really be removed in most cases.
Eg. CD-burning example: forcefully taking the resource away from a process may lead to undesirable results.
Deadlock prevention

#4 Circular wait can be attacked in different ways:

- Allow processes to hold only one resource at a time. Simple solution, but not a good one.
- Number resources and require allocation to follow a certain order: More complex, and not always a desirable solution. Inventing an ideal ordering that will work will for all processes is hard. However it solves the problem since it forces "progress" in a certain direction in the resource/process graph, so no cycles can be formed.
Ignoring potential deadlocks

As mentioned, one way of "dealing" with deadlocks is to simply not deal with them. If there is a 1/50000 chance of the system crashing due to a deadlock every day, but a 1/100 chance of crashing due to another problem, then why implement an algorithm that will prevent deadlocks but reduce the system's speed by eg. 20%?

A UNIX, Linux, Minix, etc. OS kernel typically does not concern itself with potential IO deadlocks caused by user-space processes. It leaves that responsibility up to the actual user-space programs (or the user) to deal with.

- Will you start a CD burning program and then start another program that will require the CD, and expect everything to work? Likely not.
- Will you write a client/server program pair that may encounter deadlock without adding code to detect and deal with deadlocks?
- If a some processes (eg. a game) fail to start because of a lack of system resources (eg. memory) do A) blame the OS for not implementing a sophisticated scheduler B) buy more RAM?
Deadlock avoidance

There are algorithms for managing resources in such a way that deadlocks are always avoided. Algorithms discussed in the book are:

- The banker's algorithm for one type of resource (p 248)
- The banker's algorithm for multiple resource types (p 250)
- Resource trajectories (p 249)

These algorithms solve the problem, but all have a major flaw: They require information that may not practically be available (what resources a process will require at what time is not usually easy – if at all possible – to determine ahead of time)
Resource trajectories
Banker's algorithm

When the OS gets a request for a resource from a process, it first checks whether granting that resource would lead to a safe state or not. If so, OS grants the request. Otherwise OS suspends that process until others have released enough resources.