Deadlock

CISC 3320 — CUNY Brooklyn College
Lecture Notes

Deadlock

In this chapter, we will:

1. Define what a deadlock is and learn the necessary conditions for the formation of a deadlock.
2. Illustrate situations with or without a deadlock using a resource-allocation graph.
3. Learn how to detect, prevent, and avoid deadlock situations.

Deadlock

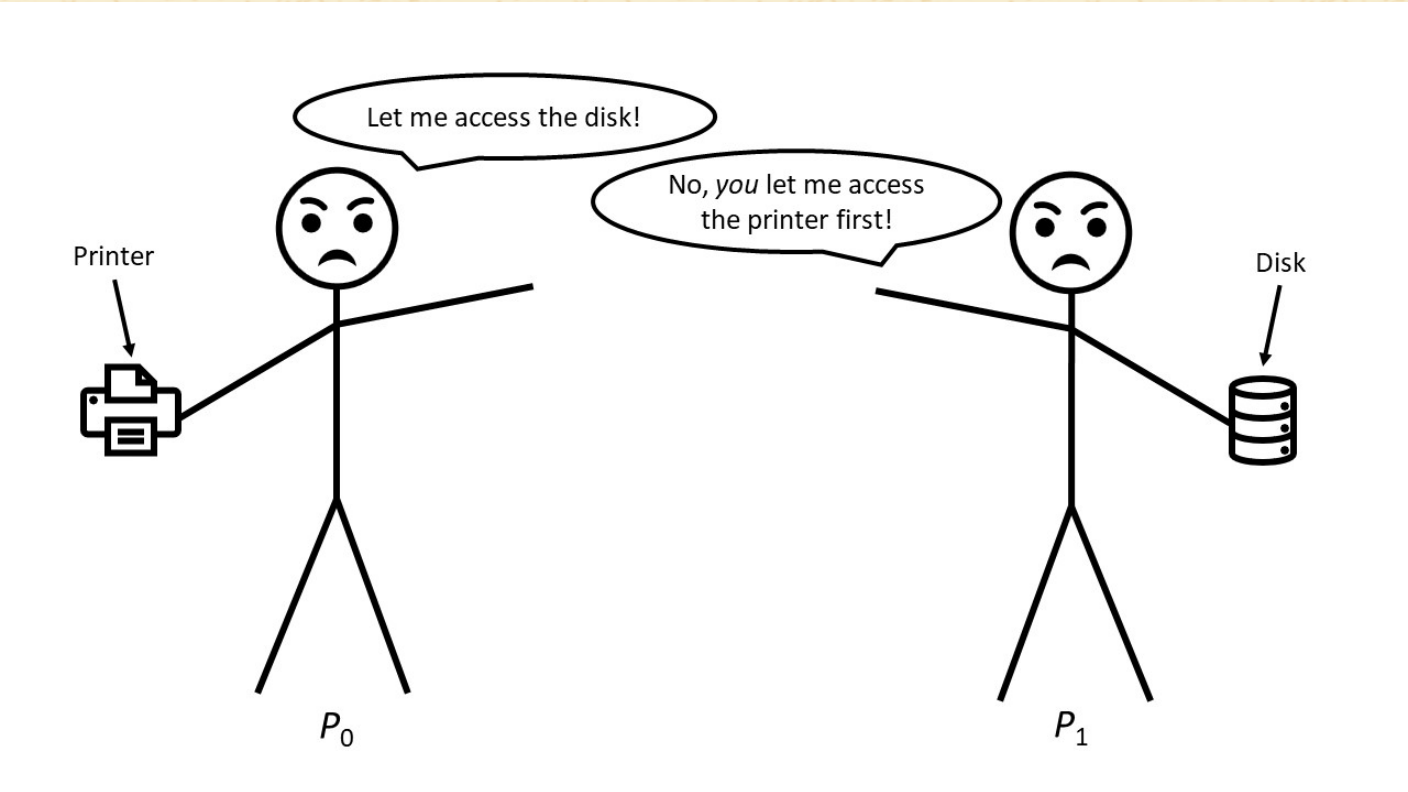
Suppose that processes P_0 and P_1 are executing. Process P_0 is accessing and holding the lock on the computer's printer, and process P_1 is accessing and holding the lock on the disk. P_0 will only release the printer if it gets access to the disk, and P_1 will only release the disk if it gets access to the printer. This is an example of a deadlock.

A **deadlock** is a situation in which two or more processes or threads are waiting forever for the resources that other processes hold and refuse to release until they get their friends' resources.

Resources include memory frames, printers, CPUs, open files, tape drives, CD-ROMS, etc., and are usually limited in amount.

Sometimes there are several resources of the same type, like 2 printers, and it doesn't matter for a process which printer to use. Still, in many cases, there may be only one resource of a type (e.g., a single CPU.)

Deadlock



Example of a Deadlock on Two Processes. Miriam Briskman, CC BY-ND 4.0.

Deadlock

There are four conditions that are necessary for a deadlock to exist. If even one of these conditions isn't true, there is no deadlock:

1. **Mutual Exclusion**: At least one resource is held in a non-sharable mode by some process/thread. If any other process requests this resource, then this 2nd process must wait for the resource to be released by the holder.
2. **Hold and Wait**: A process must be simultaneously holding at least one resource and waiting for at least one resource that is currently being held by some other process.
3. **No preemption**: Once a process is holding a resource (i.e. once its request has been granted), that resource cannot be taken away from that process until the process voluntarily releases it. In other words, the OS won't forcefully take a resource held by any process from it.
4. **Circular Wait**: There exists a group of processes that are waiting for each other's resources. Mathematically, the group of processes is $\{P_0, P_1, P_2, \ldots, P_N\}$, and each process P_i is waiting for the resource that P_{i+1} holds.

   (Note: Circular Wait implies that Hold and Wait is true, but it is easier to consider the four conditions separately.)

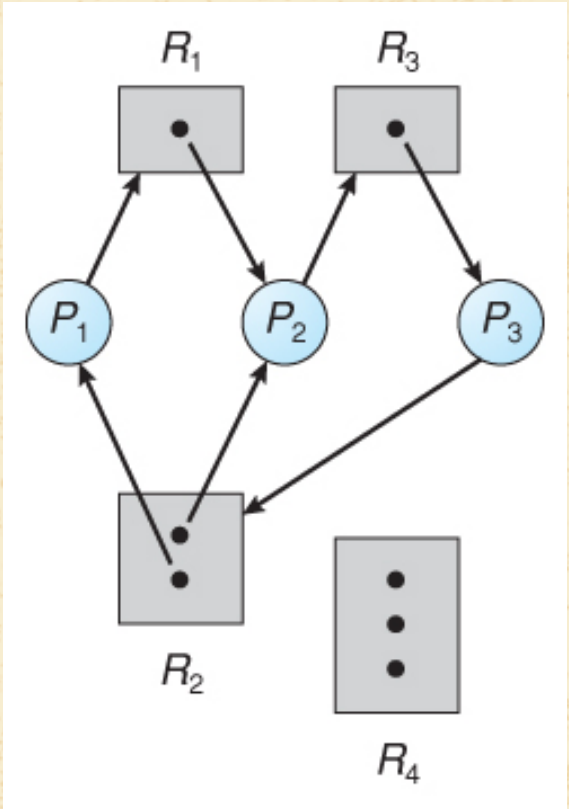
Deadlock

Using a diagram called the **Resource-Allocation graph**, we can, in may cases, learn if a deadlock occurs.

This graph contains the following components:

1. **Resource categories**: $\{R_0, R_1, R_2, \ldots, R_M\}$, which we show on the graph using square box nodes. Each box will contain at least one dot to indicate specific instances of a resource. For instance, two dots might represent two laser printers.)
2. **Processes**: $\{P_0, P_1, P_2, \ldots, P_N\}$. The # of processes might be different from this of resource categories.
3. **Request arrows**. An arrow from some process P_i to some resource category R_j indicates that P_i wants to use an instance of R_j. It doesn't matter which instance will be used since any instance of this resource will do.
4. **Assignment arrows**. An arrow from an instance (dot) of some resource R_j to some process P_i tells that this specific instance of R_j is currently held by process P_i.

Deadlock



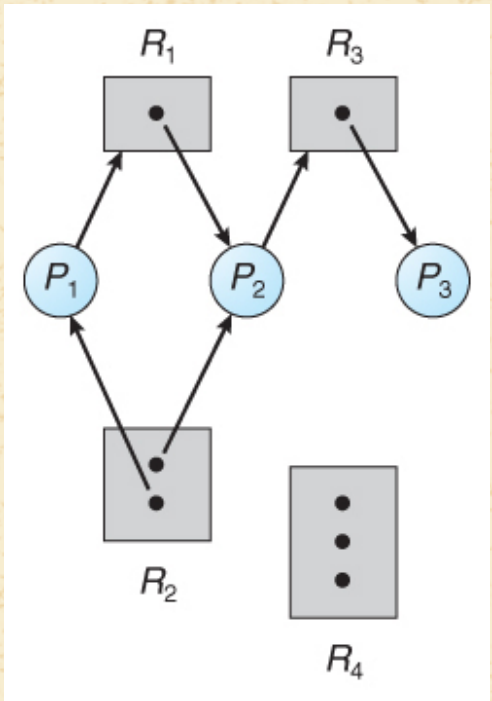
Resource-allocation graph with a deadlock. Taken from Bell, John T. "Deadlocks." University of Illinois, Chicago.

Deadlock

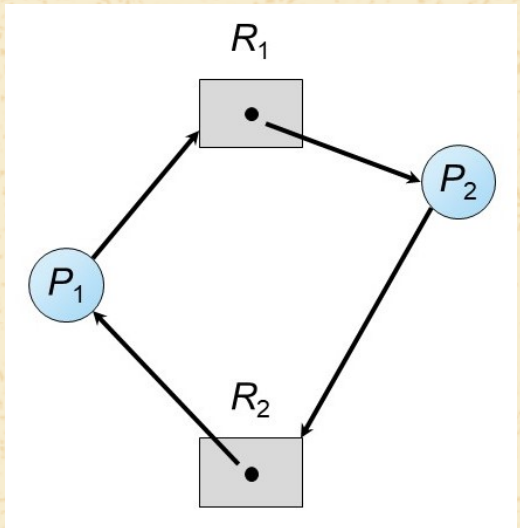
Here are some rules for possibly detecting deadlocks using the resource-allocation graph:

1. Case 1: If the graph has no cycles, then there is **no** deadlock.
2. Case 2: Otherwise, if the graph has a cycle **AND** there is one resource instance (dot) in each of the resource categories in the cycle, then there **is** at least one deadlock.
3. Case 3: Otherwise, if there is a cycle, but at least one category contains 2 or more resource instances, there **may or may not** be a deadlock. To tell exactly, we must look at the particular situation to see if it will result in a deadlock.

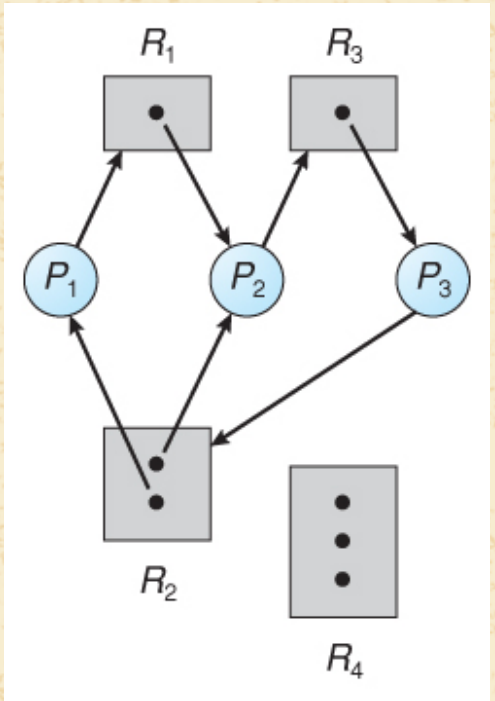
Deadlock



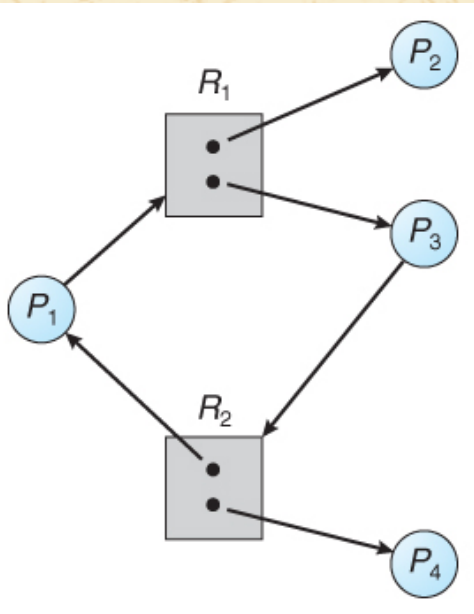
Case 1: No Cycles. Taken from Bell, John T. "Deadlocks." University of Illinois, Chicago.



Case 2: A Cycle and a Deadlock. Miriam Briskman, CC BY-ND 4.0.



Case 3: A Cycle and a Deadlock. Taken from Bell, John T. "Deadlocks." University of Illinois, Chicago.



Case 3: A Cycle but No Deadlock. Taken from Bell, John T. "Deadlocks." University of Illinois, Chicago.

Deadlock

Any questions?

Handling Deadlocks

We can deal with the deadlock problem in one of three ways:

1. We can use rules to **prevent** or **avoid** deadlocks, ensuring that the operating system will *never* enter a deadlocked state.
2. We can allow the OS to enter a deadlocked state, **detect** it, and then **recover** from it.
3. We can **ignore** the problem altogether and pretend that deadlocks never occur in the system. Surprisingly, this is the approach Windows and Linux systems take, as most deadlocks occur rarely, and it is better to let them happen and restart the computer instead of letting the operating system spend time on constantly checking for the existence of deadlocks and recovering from them.

To *prevent* deadlocks, the OS could prevent one of the four necessary conditions that make a deadlock occur. To *avoid* deadlocks, the OS needs to know more details about the running processes and the resources that they will request to use. Deadlock detection is fairly straightforward, but deadlock recovery requires either aborting processes or preempting resources, neither of which is usually desired.

Handling Deadlocks: Prevention

To prevent deadlocks, the OS could prevent one of the four necessary conditions for a deadlock:

1. **Mutual Exclusion**: Unfortunately, this condition can't be prevented and must always hold to avoid race conditions. However, note that resources that can be shared by several processes, like read-only files, don't cause deadlocks.
2. **Hold and Wait**: The OS must prevent processes from holding one or more resources while simultaneously waiting for one or more others. This could be done by requiring that processes holding resources release them before requesting new resources.
3. **No preemption**: The OS must do the following: if a process is forced to wait when requesting a new resource, then all other resources previously held by this process are implicitly released (preempted), forcing this process to re-acquire the old resources along with the new resources in a single request, similar to the previous solution.
4. **Circular Wait**: The OS must enumerate all the resources and then require that processes request resources only in strictly increasing (or decreasing) order. In other words, in order to request resource R_j, a process must first release all the resources $\{R_0, R_1, R_2, \ldots, R_{j-1}\}$ that it holds.

Handling Deadlocks: Avoidance

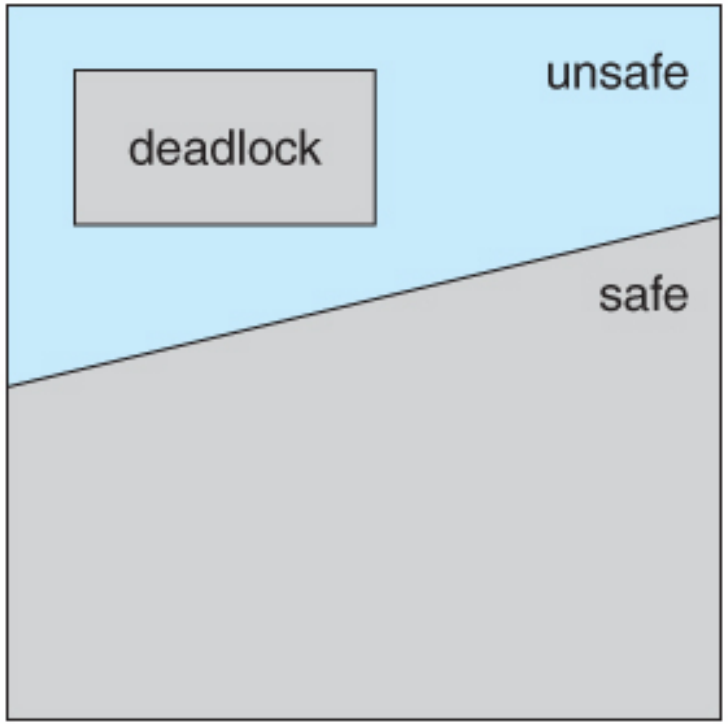
To avoid deadlocks, the OS is required to know more information about each process, such as the number of available resources of each type and the maximum number of resources that a process could request during execution. Keeping track of this info leads to low device utilization.

When a scheduler sees that starting a process or granting resource requests *may* lead to future deadlocks, then that process is simply not run by the scheduler, or the request is not granted, a decision that avoids deadlocks.

A **resource allocation state** is a table containing the number of allocated resources, the number of available resources, and the number of maximum potentially required resources per each of the running processes.

A resource allocation state is called **safe** when the system can allocate all resources requested by all processes (up to their stated maximums) without entering a deadlock state. If a safe sequence does not exist, then the system is in an **unsafe** state, which *may* lead to deadlock. All safe states are deadlock free, but not all unsafe states lead to deadlocks.

Handling Deadlocks: Avoidance



Safe, unsafe, and deadlocked state spaces. Taken from Bell, John T. "Deadlocks." University of Illinois, Chicago.

Handling Deadlocks: Avoidance

Consider the following state with 3 processes and 12 disks. We avoid deadlock by following the steps below the table:

Process	Maximum Needed:	Currently Allocated:	Available:
P_0	10	5	3
P_1	4	2	
P_2	9	2	

1. Process P_1 needs to access a total of 4 disks, 2 of which it already accesses. Since there are 3 available disks, we can safely let P_1 access 2 more disks. After P_1 finishes accessing all 4 disks, there will be 2 + 2 + 1 = 5 available disks.
2. Next, notice that process P_0 needs to access a total of 10 disks, 5 of which it already accesses. Since there are now 5 available disks, we can safely let P_0 access these 5 available disks. After P_0 finishes accessing all 10 disks, there will be 10 available disks.
3. Finally, process P_2 needs to access a total of 9 disks, 2 of which it already accesses. Since there are now 10 available disks, we can safely let P_2 access 7 of the available disks. After P_2 finishes accessing all 9 disks, there will be 12 available disks, so we avoid deadlocks!

Handling Deadlocks: Avoidance

Consider a slightly different state with 3 processes and 12 disks:

Process	Maximum Needed:	Currently Allocated:	Available:
P_0	10	5	2
P_1	4	2	
P_2	9	3	

Although process P_1 can safely use 2 of the available disks to finish its work, the 4 disks that will become available after P_1 finishes executing won't satisfy the requests of neither of P_0 nor P_2 since 10 - 5 = 5 > 4 and 9 - 3 = 6 > 4, so there are not enough available disks for these requests.

In other words, this is an unsafe state. This, however, doesn't mean that we will get a deadlock, as some of the processes might release the used resources voluntarily, which will increase the number of available disks, and therefore never cause a deadlock.

Deadlock

Any questions?

Deadlock: Sources

Bell, John T. "Deadlocks." *University of Illinois, Chicago*, Accessed 11 July 2022. URL: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/7_Deadlocks.html

Chapter 4: Synchronization and Deadlocks (pp. 124-129).

Hailperin, Max (2019). Operating Systems and Middleware: Supporting Controlled Interaction. (Revised edition 1.3.1) S.l.: Gustavus Adolphus College. This work is licensed under the Creative Commons Attribution-ShareAlike3.0 Unported License. URL: https://gustavus.edu/mcs/max/os-book/osm-rev1.3.1.pdf#chapter.4