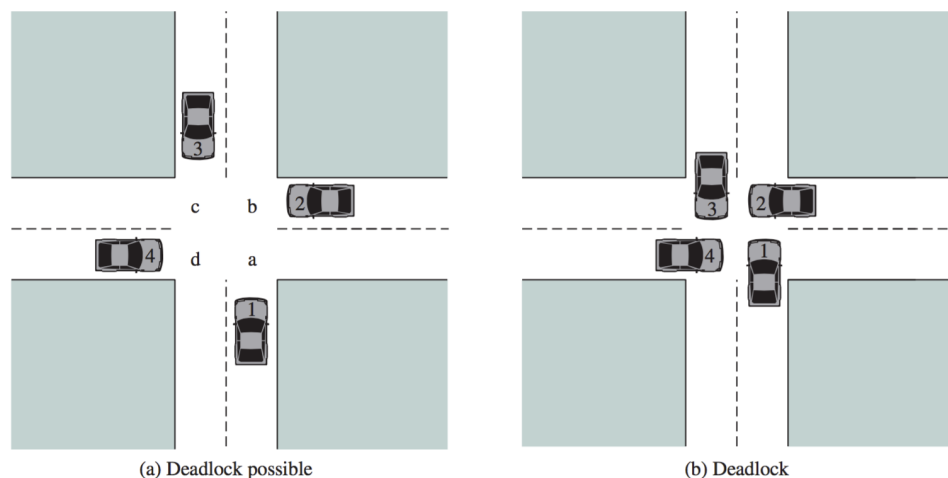# Deadlock

We have already introduced the subject of deadlock and gave an informal definition as all processes being "stuck" (unable to proceed). A more formal definition is given in [Sta14]: "the *permanent* blocking of a set of processes that either compete for system resources or communicate with each other". There is emphasis on permanent. It may be possible for all processes to be stuck temporarily, because one is waiting for some event (e.g., a read from disk), but this situation will resolve itself and is not deadlock. A set of processes is truly deadlocked when each process in the set is blocked on some event that can only be triggered by another blocked process in the set. In this case it is permanent, because none of the events can take place.

A deadlock involves some conflicting needs for resources by two or more processes. Consider a traffic deadlock. Suppose four cars arrive at a four-way stop intersection at the same time. We can divide the intersection into four quadrants, labelled, a, b, c, and d. To drive straight through the intersection, any one vehicle will need at least two of the quadrants (the two directly in front of it).

An illustration of deadlock involving cars [Sta14].

According to the Highway Traffic Act, whichever vehicle arrives at the stop sign first has the right of way. In the event that two vehicles arrive at the same time, a vehicle yields the right of way to the vehicle on its right. As long as three or fewer vehicles come to the stop sign at the same time, this works. If all four vehicles arrive at the same time, we have the potential for a problem. It is not a deadlock yet, because none of the processes are stuck yet, but it could happen. If all the drivers believe they should go first, we get the situation on the right, and we actually do have deadlock. This is very much like the dining philosophers problem; deadlock occurs if everyone tries to do the same thing at the same time.

Of course, for deadlock to occur, we do not have to have symmetric processes trying to do the same thing at the same time. Given two semaphores, a and b, and two processes, we can have the following code that will sometimes, but not always lead to deadlock. If thread P locks a and then there is a process switch, and b is locked by Q, both threads will be stuck. Each has one resource the other needs, but they are both blocked and waiting for the other. It is obvious when we compare threads P and Q like this side-by-side, but in reality the problem is not usually that easy to see.
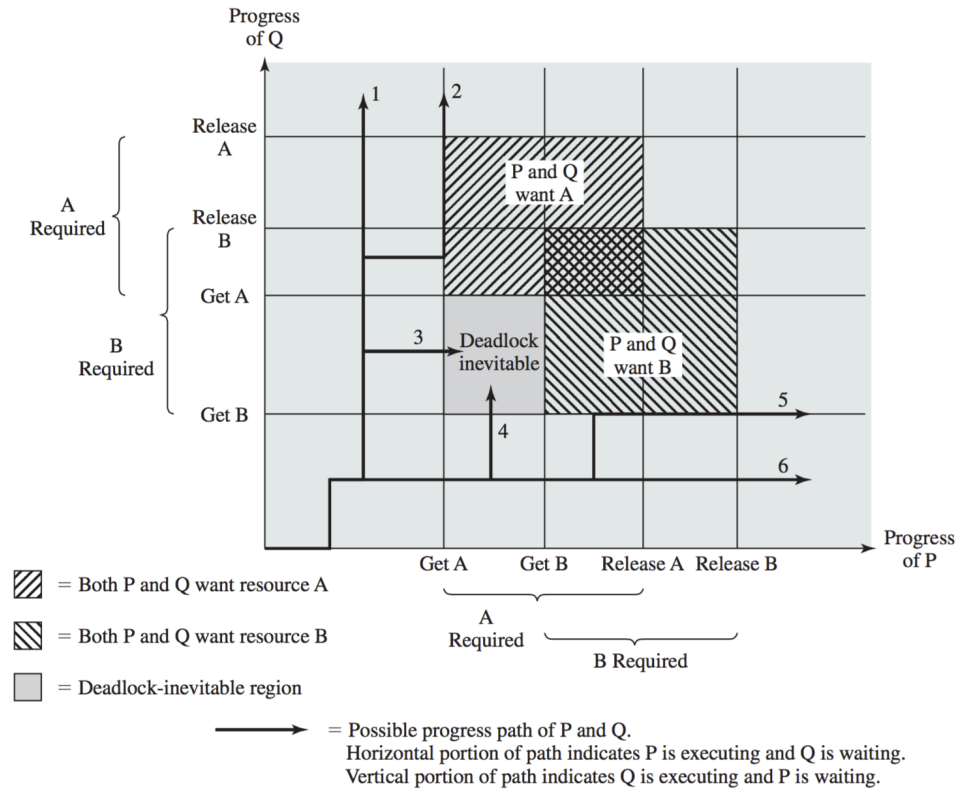
Thread P
```
1. wait( a )
2. wait( b )
3. [critical section]
4. signal( a )
5. signal( b )
```

Thread Q
```
1. wait( b )
2. wait( a )
3. [critical section]
4. signal( b )
5. signal( a )
```

We can represent the code with a *joint progress diagram*, a visual representation of the advancement of the threads:

A joint progress diagram representing the behaviour of the P and Q code [Sta14].

In this diagram there are six different possible execution paths, labelled 1-6, as follows, from [Sta14]:

1. Q acquires B and then A, then releases B and A. Then P acquires both resources and releases them.

2. Q acquires B and then A. P executes and blocks on a request for A. Q releases B and A. P is unblocked and continues, acquiring both resources and releasing them.

3. Q acquires B and then P acquires A. Deadlock is inevitable.

4. P acquires A and then Q acquires B. Deadlock is inevitable.

5. P acquires A and then B. Q executes and blocks on a request for B. P releases B and A. Q is unblocked and continues, acquiring both resources and releasing them.

6. P acquires A and then B, then releases A and B. Then Q acquires both resources and releases them.

The area of the diagram where it says "Deadlock Inevitable" is known as the *fatal region*. The existence of a fatal region means that deadlock is inevitable (will eventually happen), but any one particular execution of the program may or may not result in deadlock. The joint progress diagram works will when there are two threads sharing resources, but for a situation of n processes sharing resources, an n-dimensional diagram would be needed. For $n \geq 4$, I at least can't represent it visually on a 2D surface, and it has nothing to do with my poor art skills.

Reusable and Consumable Resources

So, deadlock takes place when two processes or threads are competing for resources. We can generally classify a resource as either *reusable* or *consumable*. A reusable resource can be used by one process at a time, and is not depleted by that use. A process may lock the resource, make use of it, then release it such that other processes may acquire it. Processors, memory, files, and semaphores are all examples of reusable resources. If process P gets resource A and then releases it, process Q can acquire it. Thus, the example above involving P and Q is a deadlock involving reusable resources.

A consumable resource is one that is created and destroyed upon consumption. If the user presses the "Z" key on the keyboard, this generates an interrupt and produces the "Z" character in a buffer. A process that takes input will then consume that character (e.g., it goes into the vi editor window) and it is unavailable to other processes. Other things that are consumable resources: interrupts, signals, and messages. Deadlock is somewhat less likely when dealing with consumable resources, but in theory, all processes could be waiting to receive a message, if it is a blocking receive, and no process can send a message (because they are all waiting for some other process to send first).

Conditions for Deadlock

When a disaster happens, it is typically a result of a chain of things going wrong. If any one of those things did not happen, the disaster would be averted. This is referred to as "breaking the chain". There are four conditions for deadlock:

1. **Mutual Exclusion**: A resource belongs to, at most, one process at a time.

2. **Hold-and-Wait**: A process that is currently holding some resources may request additional resources and may be forced to wait for them.

3. **No Preemption**: A resource cannot be "taken" from the process that holds it; only the process currently holding that resource may release it.

4. **Circular-Wait**: A cycle in the resource allocation graph.

If the first three conditions are true, deadlock is possible, but deadlock will only happen if the fourth condition is fulfilled. But what is a resource allocation graph? It is a directed graph that tells us the state of the system by representing the processes, the resources, and which resources are held by which processes. Consider the example below:
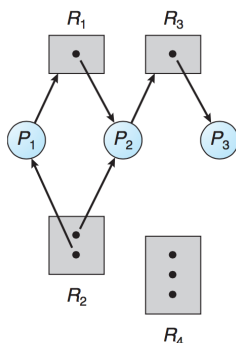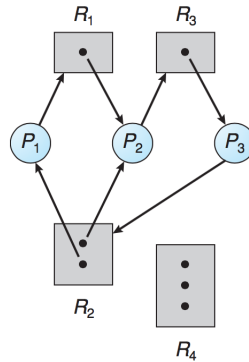

A sample resource allocation graph [SGG13].

A process is represented by a circle labelled P_n. A resource is a box labelled R_n and contains one or more black circles, representing how many of that resource are available. A binary semaphore would therefore have one black circle; a general semaphore will have 1 or more. A directed edge from P_i to R_j is a request: a process requests

that resource. A directed edge from R_j to P_i represents assignment - the process holds that resource. When a request is made, a request edge is inserted into the graph; if the request is fulfilled it is instantly turned into an assignment edge. When a resource is released, the assignment edge is deleted [SGG13].

If there are no cycles in the graph, then we can be certain that no process in the system is deadlocked. If a cycle exists, then some process may be deadlocked:

A sample resource allocation graph with a deadlock [SGG13].

But the presence of a cycle, on its own, is not necessarily certain that there is a deadlock:
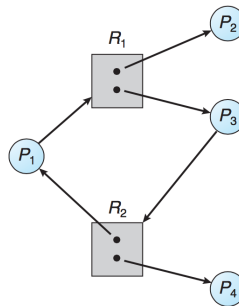
A sample resource allocation graph with a cycle, but no deadlock [SGG13].

In this case, P_4 and P_2 are not part of the cycle and when they release their held resources, the other two processes will be able to proceed.

Dealing with Deadlock

There are four basic approaches to dealing with deadlock, each of which we will examine in turn.

1. Ignore it.
2. Deadlock prevention.
3. Deadlock avoidance.
4. Deadlock detection.

Deadlock Option 1: Ignore it

This option is certainly convenient for operating system designers: we simply pretend that deadlock can never happen, or if it does happen, it is the application developers' fault. While it is tempting and easy to just define a problem as being nothing we need to deal with, that's unrealistic in reality. That said, this is the approach taken in Microsoft Windows, so it is a valid option. If two processes get deadlocked, the user may simply see no progress (or the "not responding" dialog) and have to open task manager and start killing processes.

Setting aside the "do-nothing" option, let us move on to some approaches that actually deal with the problem.

Deadlock Option 2: Deadlock Prevention

This approach is a way of preventing a deadlock from being possible. The first three conditions for deadlock (mutual exclusion, hold and wait, and no preemption) are all necessary for deadlock to be possible. If we eliminate one of these three pillars, deadlock is not possible and it is prevented from happening.

Mutual Exclusion. This pillar cannot, generally speaking, be disallowed. The purpose of mutual exclusion is to prevent errors like inconsistent state or crashes. Getting rid of mutual exclusion to rule out the possibility of deadlock is a cure that is worse than the disease. It is therefore not acceptable as a solution.

Hold and Wait. To prevent the hold-and-wait condition, we must guarantee that when a process requests a resource, it does not have any other resource. In a practical sense this does not mean that things can be requested only one at a time; that would be like telling the philosophers that when they have one chopstick already, they cannot request another.

One plausible solution is that the process must request all resources at the beginning of the program. So if the program is going to need resources R_1, R_2, and R_3 at some point in the program, all three must be requested right at the beginning and held throughout the program. No further resources may be requested at any time during execution. Suppose the program does a lot of processing and then prints the result on a printer (and the printer is R_3): the printer is tied up by this process throughout the entire execution of the program [SGG13].

Unfortunately, this requires some amount of clairvoyance: a process has to know in advance all of the resources that it will need. Remember that a file is a resource, and even a simple program like a text editor can be used by the user to open an arbitrary file; how do we know in advance which will be requested?

This also has performance implications: a process cannot start until it has all the resources it will ever need, even if it will not need them until much later. Thus, processes might spend a lot of time waiting before starting; something that users may not tolerate. In theory, a process might never start if one or more of the resources it needs is always in use (so this is vulnerable to starvation).

An alternative is that a process must release all its currently-held resources before it can get any new ones. So, if the process has R_1 and R_2 and wants to get R_3 (and needs all three), the process must first release R_1 and R_2 before it can request all three of them.

Unfortunately, this runs into some problems, too. A resource that cannot be easily released is memory, as released memory may be collected and reassigned by the operating system. Therefore we cannot release all resources, and releasing only some of the resources means that we cannot categorically rule out deadlock; we can only make it less likely to occur [HZM14].

Another idea that might work is *two-phase locking*. A process attempts to lock a group of resources at once, and if it does not get everything it needs, it releases the locks it received and tries again. Thus a process does not wait while holding resources. If a philosopher picks up a chopstick but is unable to acquire a second, she puts down the chopstick she has picked up and tries again. Although she may not be the one to eat next, at least not all philosophers are stuck holding one chopstick.

Two phase locking is not applicable to our current model for semaphores, where there is no way to know the value

of the semaphore and the operating system will block a process on a `wait` if some other thread is in the critical region. After the process is blocked on the semaphore, a second process will run, and the first process does not get the opportunity to release the resources it holds. With that in mind, there are systems that support nonblocking requests for resources and mutual exclusion areas that return, perhaps, `true` if the resource is acquired and `false` if it is unavailable. Then the program is responsible for checking if any of the requests returned `false` and releasing any resources where the request returned `true`.

The idea of a process releasing resources that it is holding is the basis for the next solution: knocking down the "no preemption" pillar.

No Preemption. If we violate this condition, it means that we do have preemption: forcible removal of resources from a process. Suppose a process P_1 holds R_1 and R_2 and wants to get R_3, but R_3 is unavailable. P_1 will be blocked by the operating system. If another process P_2 comes by and requests R_1 and R_2, the resources R_1 and R_2 are taken away from P_1. The resources are added to the list of things that P_1 is waiting for (so it is waiting for all three now). In the meantime, P_2 can use them and continue. P_1 will be unblocked when all three resources are once again available for it to acquire [SGG13].

For preemption to work, however, the resource must be a resource of a type where the state can be saved and restored (e.g., the CPU with its registers). This is not applicable to all resources; if a printer is in use by P_1 it cannot be preempted and given to P_2, otherwise the printout will be a jumble. Thus, preemption is also not sufficient to prevent deadlock from ever transpiring, it once again only makes it less likely.

References

[HZM14] Douglas Wilhelm Harder, Jeff Zarnett, and Vajih Montaghami. *A Practical Introduction to Real-Time Systems for Undergraduate Engineering*. 2014. Online; version 0.14.12.22.

[SGG13] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts (9th Edition)*. John Wiley & Sons, 2013.

[Sta14] William Stallings. *Operating Systems Internals and Design Principles (8th Edition)*. Prentice Hall, 2014.