The Dining Philosophers Problem

The dining philosophers problem was also proposed by Dijkstra in 1965. The problem can be formulated with n philosophers, but problem is typically described as involving five philosophers. The description that follows is from [SGG13]. These five smart individuals spend their lives thinking, but every so often, they need to eat. They share a table, each having his or her own chair. In the centre of the table is a bowl of rice, and the table is laid with five single chopsticks. See the diagram below.
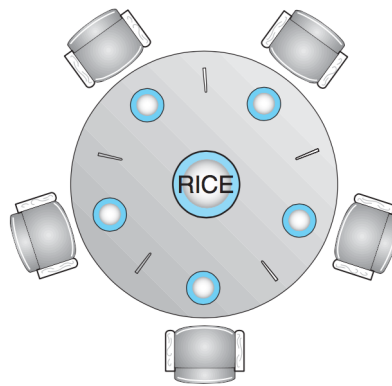
The situation of the dining philosophers [SGG13].

When a philosopher wishes to eat, she sits down at her designated chair, and attempts to pick up the two chopsticks that are nearest (one on the left, and one on the right). Philosophers are polite and therefore do not grab a chopstick out of the hands of a colleague. When a philosopher has both chopsticks, she may eat rice, and when she is finished, she puts down the chopsticks and goes back to thinking.

Some textbooks formulate this problem as philosophers needing two forks to eat rather than chopsticks. It is, of course, much easier to imagine having difficulty eating with only one chopstick than having difficulty with only one fork. Of course, the scenario is a little bit silly; we don't study it because it is supposed to be a true to life model of how real philosophers behave. The scenario is just a convenient and memorable example of a whole class of problems.

Suppose then that semaphores are the method for managing things. Because only one person can be in possession of a chopstick at a time, each chopstick may be represented by a binary semaphore. As a philosopher needs the chopstick to his left and right to eat, when the philosopher sits down he attempts to acquire the left chopstick, then the right, eats, and puts the chopsticks down. This works fine, until all philosophers sit down at the same time. Each grabs the chopstick to his or her left. None of them are able to acquire the chopstick to his or her right (because someone has already picked it up). None of the philosophers can eat; they are all stuck. This is deadlock.

This example makes it more clear why we call a situation where a thread never gets to run "starvation". If a philosopher is never able to get both chopsticks, that philosopher will never be able to eat, and though I am not an expert on biology, I have it on good authority that people who do not eat anything end up eventually starving to death. Even philosophers.

One thing that would guarantee that this problem does not occur is to protect the table with a binary semaphore. This would allow exactly one philosopher at a time to eat, but at the very least, deadlock and starvation would be

avoided. Although this works, it is a suboptimal solution. There are five seats and five chopsticks and yet only one person is eating at a time. We can get better concurrency and use of the resources.

Next idea: what if we limit the number of philosophers at the table concurrently to four? The pigeonhole principle applies here: if there are k pigeonholes and more than k pigeons, at least one pigeonhole must have at least two pigeons. Thus, at least one of the four philosophers can get two chopsticks [HZMG15]. Implementing the solution is easy; we have a general semaphore with a maximum and initial value of 4.

Another idea: the problem above occurs because every philosopher tries to pick up the left chopstick first. If some of them try to pick up the left and some pick up the right first, then deadlock will not happen, either [SGG13].

The dining philosophers problem is a good springboard from which to launch into a much deeper discussion of deadlock, and starvation.

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).
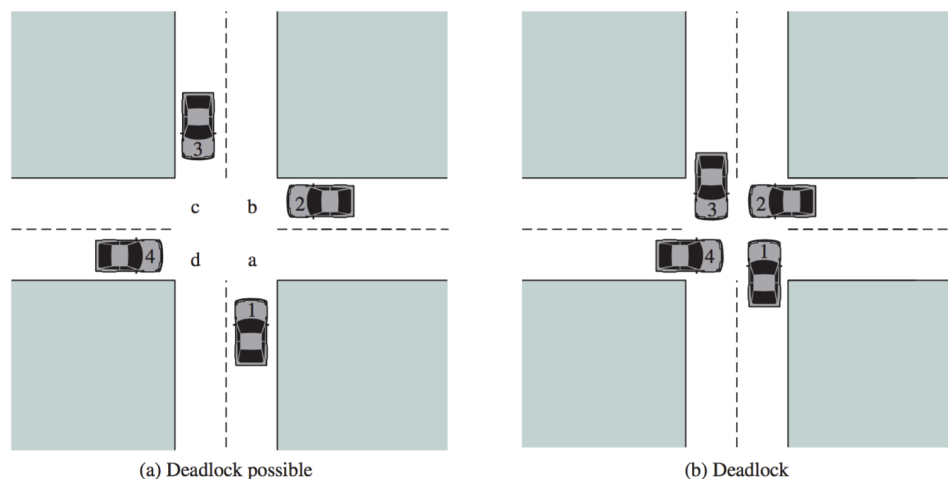
An illustration of deadlock involving cars [Sta14].

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. post( a )
5. post( b )
```

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

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, posts, 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:
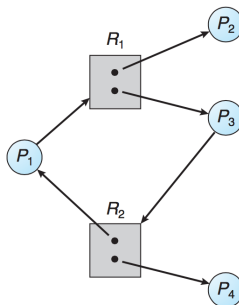
A sample resource allocation graph [SGG13].

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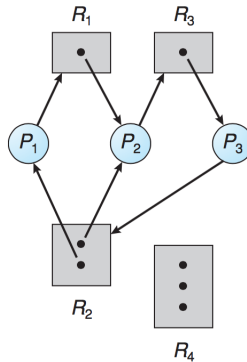
A sample resource allocation graph with a deadlock [SGG13].

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: we simply pretend that deadlock can never happen, or if it does happen, it is someone else's fault. 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. But the operating system designers can plausibly blame the problem on the program authors. Since we are the program authors we can't really pass the buck here. You can try to blame the users, but they'll tell you that your system shouldn't allow them to make mistakes in the first place. Well then.

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.

When it comes to your specific program, though, you might be able to make it so that mutual exclusion is not necessary. If you can give each thread, for example, its own copy of the data, or you can find an algorithm that no longer requires locking a given resource, then by all means, use it! Locking and other mutual exclusion constructs are things we use because we must, not because we like to.

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.

The operating system could operate under these rules, but we know that our usual commercial operating systems don't do this. We could imagine a world where things are different, but that's not very helpful when it comes to implementing the program we're trying to write. Even so, the idea of requesting everything up front is obviously not possible (at the time of opening a document you are not asked if you are going to use the printer sometime later, just in case), nor is giving up all resources before asking for more possible. 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 [HZMG15].

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. If there exist requests for resources and mutual exclusion areas that return, perhaps, `true` if the resource is acquired and `false` if it is unavailable, this can work. Then the program is responsible for checking if any of the requests returned `false` and releasing any resources where the request returned `true`.

Fortunately, we know of some routines that do just this: the `trylock` functions that were mentioned earlier but not expanded upon:

```
int pthread_mutex_trylock( pthread_mutex_t * mutex )
int pthread_rwlock_tryrdlock( pthread_rwlock_t * rwlock )
int pthread_rwlock_trywrlock( pthread_rwlock_t  * rwlock )
```

These functions return an integer and it's extremely important to check and see if the return code is 0, because that is the only way to know if the lock was acquired. The call is non-blocking so the code will carry on regardless. Consider below a code description of the dining philosophers if they used two-phase locking via the trylock routines. Assume that the mutex variables have been initialized appropriately. It should be possible to reason about this solution and demonstrate that (1) a philosopher can only eat if they have both chopsticks, and (2) deadlock does not occur.

```
int locked_both = 0;
while( locked_both == 0 ) {
  int locked1 = pthread_mutex_trylock( chopstick1 );
  int locked2 = pthread_mutex_trylock( chopstick2 );
  if (locked1 != 0 && locked2 == 0) {
    pthread_mutex_unlock( chopstick2 );
  } else if (locked1 == 0 && locked2 != 0 ) {
    pthread_mutex_unlock( chopstick1 );
  } else if (locked1 != 0 && locked2 != 0 ) {
    /* Do nothing */
  } else {
    locked_both = 1;
  }
}
eat( );

pthread_mutex_unlock( chopstick1 );
pthread_mutex_unlock( chopstick2 );
```

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, by the operating system. 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].

This isn't really something we can do as program designers, because only the operating system really has the power to carry out. It's included for completeness, but is not anything we can actually implement ourselves.

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

[HZMG15] Douglas Wilhelm Harder, Jeff Zarnett, Vajih Montaghami, and Allyson Giannikouris. *A Practical Introduction to Real-Time Systems for Undergraduate Engineering*. 2015. Online; version 0.15.08.17.

[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.