

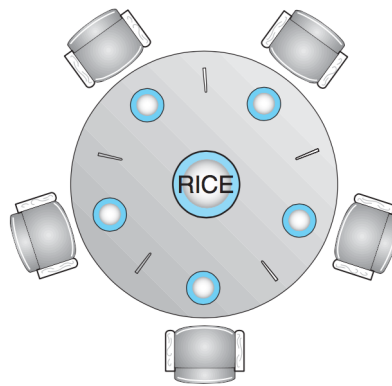
Lecture 18 — Deadlock

Jeff Zarnett

2018-08-31

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.

Humorous aside: as [HZM14] points out, 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 [HZM14]. Implementing the solution is easy; we have a general semaphore with a maximum and initial value of 4.

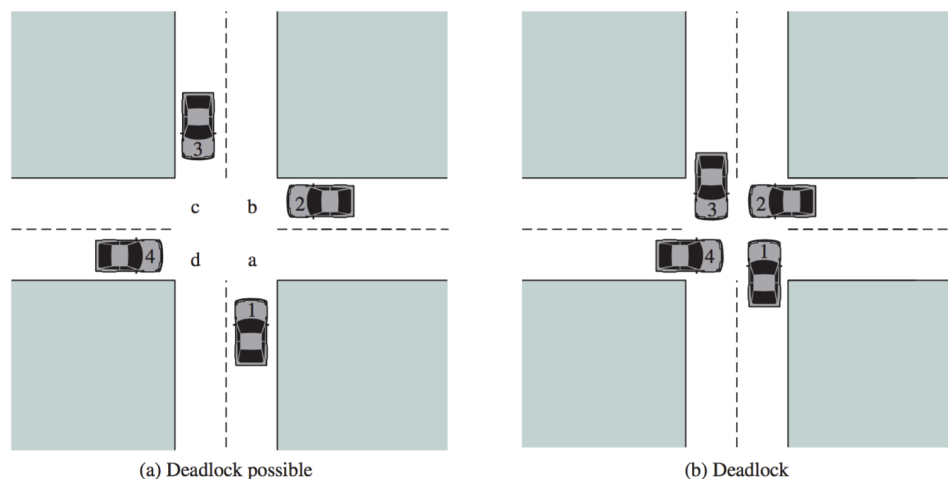
Another idea: the problem above occurs because every philosopher tries to pick up the left chopstick first. If some of them try to pick up the left and some pick up the right first, then deadlock will not happen, either [SGG13].

The dining philosophers problem is a good springboard from which to launch into a much deeper discussion of deadlock, and starvation; that is what we will examine next.

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.

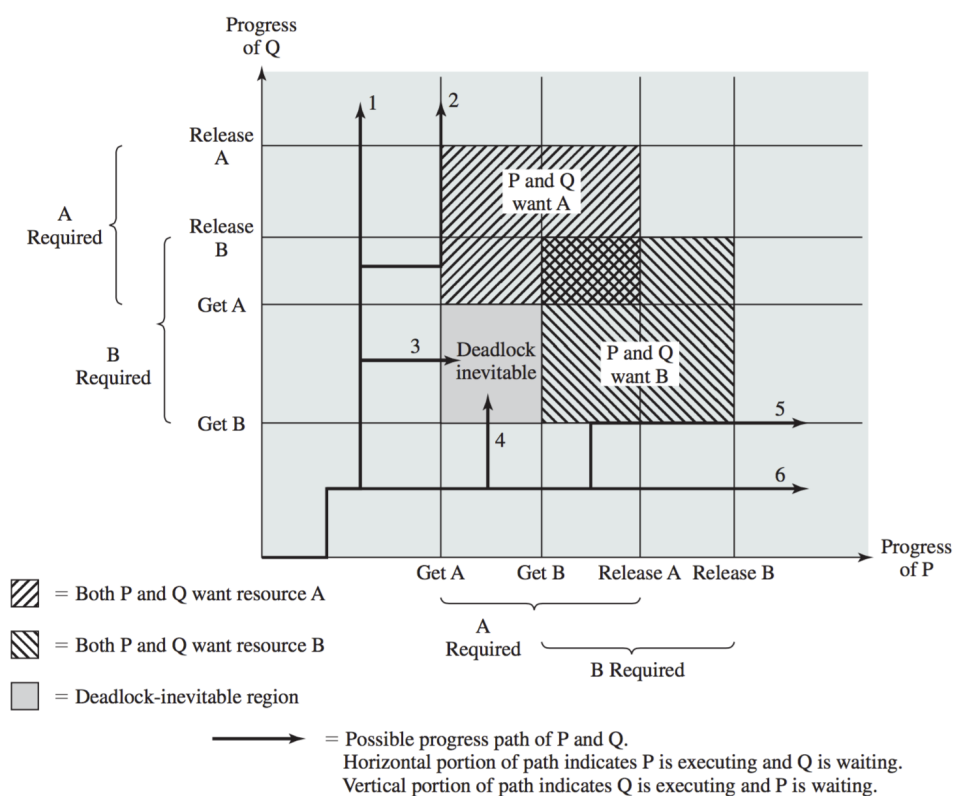
Thread P

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 well 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.

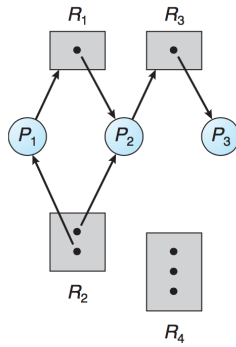
A consumable resource is one that is created and destroyed upon consumption. If the user presses the “Z” key on the keyboard, this generates an interrupt and produces the “Z” character in a buffer. A process that takes input will then consume that character (e.g., it goes into the vi editor window) and 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.

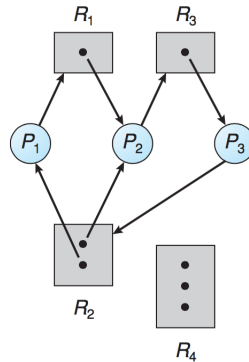
If the first three conditions are true, deadlock is possible, but deadlock will only happen if the fourth condition is fulfilled. But what is a resource allocation graph? It is a directed graph that tells us the state of the system by representing the processes, the resources, and which resources are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



A sample resource allocation graph [SGG13].

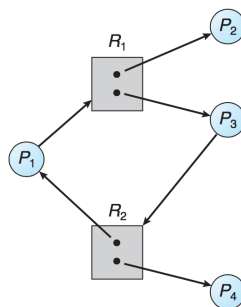
A process is represented by a circle labelled P_n . A resource is a box labelled R_n and contains one or more black circles, representing how many of that resource are available. A binary semaphore would therefore have one black circle; a general semaphore will have 1 or more. A directed edge from P_i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

References

- [HZM14] Douglas Wilhelm Harder, Jeff Zarnett, and Vajih Montaghani. *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.