Lecture 18 — Deadlock

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

September 2, 2018

The Dining Philosophers Problem

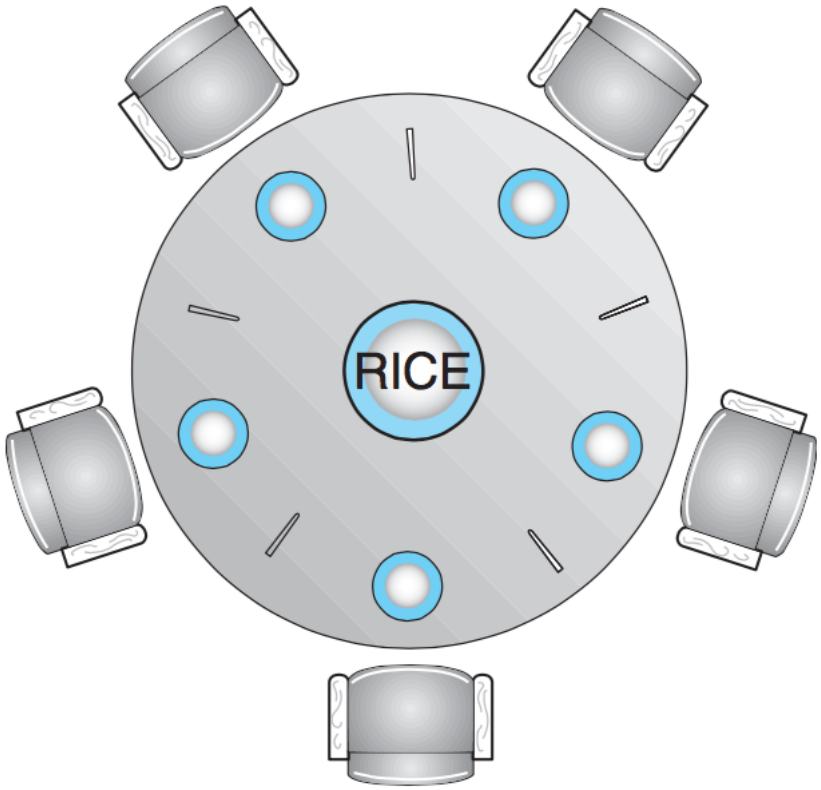
The dining philosophers problem was also proposed by Dijkstra in 1965.

The problem can have *n* philosophers, but problem is typically described as 5.

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, & it is laid with 5 single chopsticks.

The Dining Philosophers Problem



Dinner Time

When a philosopher wishes to eat, she sits down at her designated chair, and attempts to pick up the two chopsticks that are nearest.

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.

Dining with Semaphores

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.

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. Deadlock.

Deadlock at the Dinner Table

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.

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.

Saving the 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. Yet only one person is eating at a time.

No, the Pigeons are not for Eating

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.

Implementation is easy: a general semaphore with a max and initial value of 4.

Invite a Leftie

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.

This problem is a great basis to launch into a discussion about deadlock…

Deadlock

An informal definition of deadlock: all processes being “stuck”.

A more formal definition: “the *permanent* blocking of a set of processes that either compete for system resources or communicate with each other”.

It may be possible for all processes to be stuck temporarily, because one is waiting for some event (e.g., a read from disk).

This situation will resolve itself and is not deadlock.

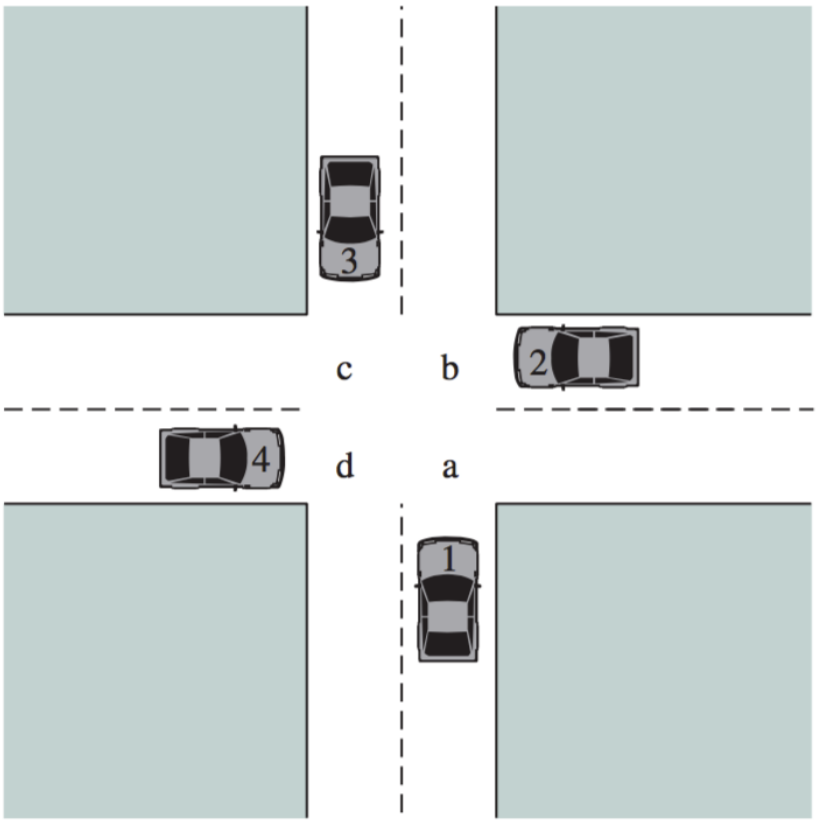
Deadlock

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.

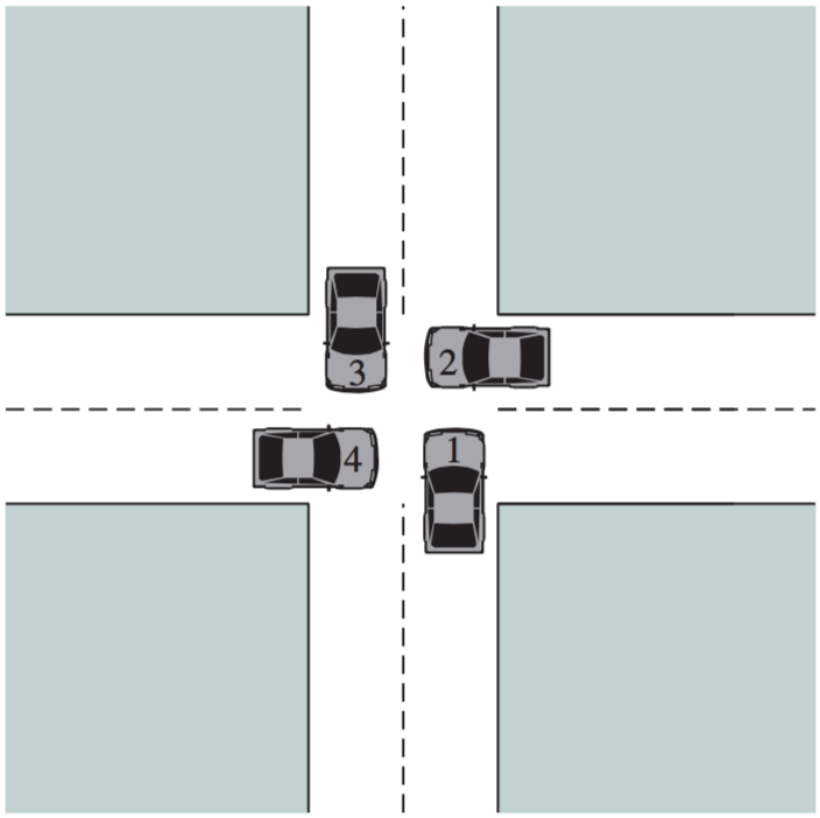
In this case it is permanent, because none of the events can take place.

A deadlock involves some conflicting needs for resources by two or more processes.

A Traffic Deadlock



(a) Deadlock possible



(b) Deadlock

A Traffic Deadlock

Whichever vehicle arrives at the stop sign first has the right of way.

If two vehicles arrive at the same time, a vehicle yields the right of way to the vehicle on its right.

Okay, as long as 3 or fewer vehicles come to the stop sign at the same time.

If all four vehicles arrive at the same time, we have the potential for a problem.

A Traffic Deadlock

It is not a deadlock yet, because none of the processes are stuck yet.

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.

Deadlocks

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.

Deadlock of *P* and *Q*

It's not always this 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 )
```

Reusable and Consumable 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.

A process may lock the resource, make use of it, then release it such that other processes may acquire it.

Processors, memory, files, semaphores are all examples of reusable resources.

Reusable and Consumable 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.

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”.

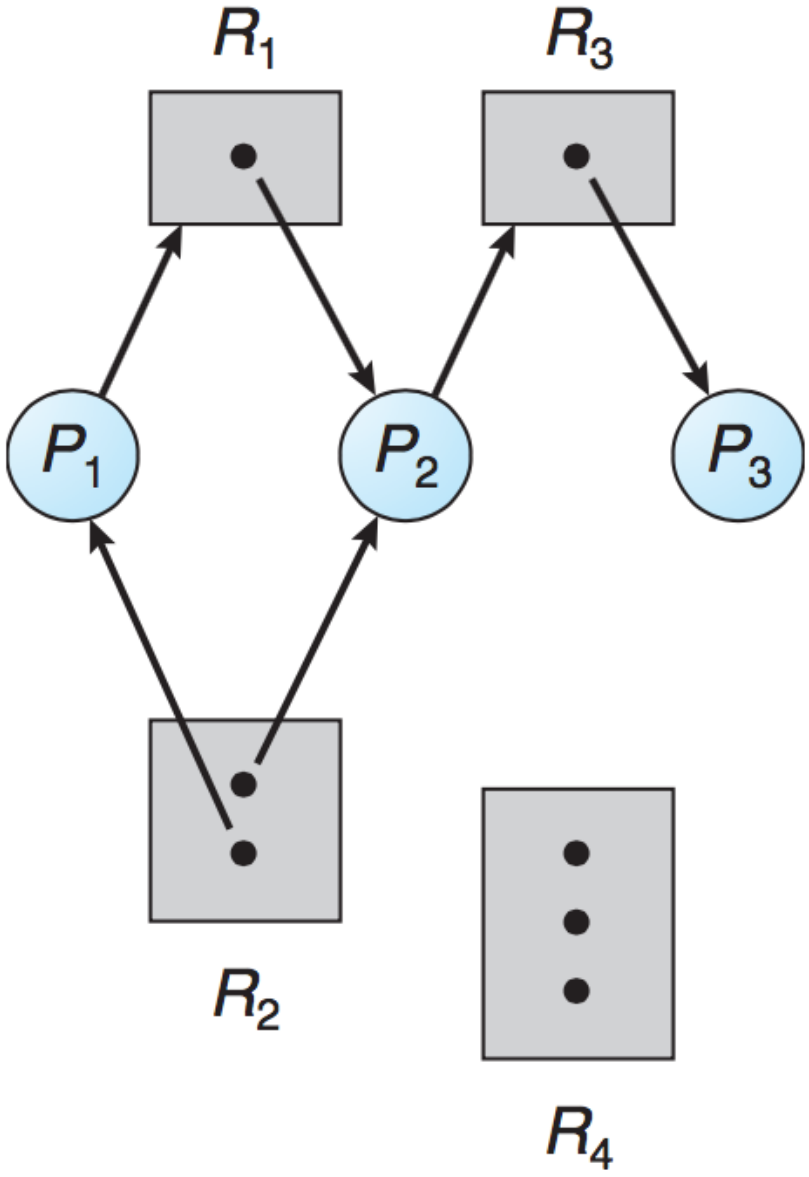
Conditions for Deadlock

There are four conditions for deadlock:

1. **Mutual Exclusion**
2. **Hold-and-Wait**
3. **No Preemption**
4. **Circular-Wait**

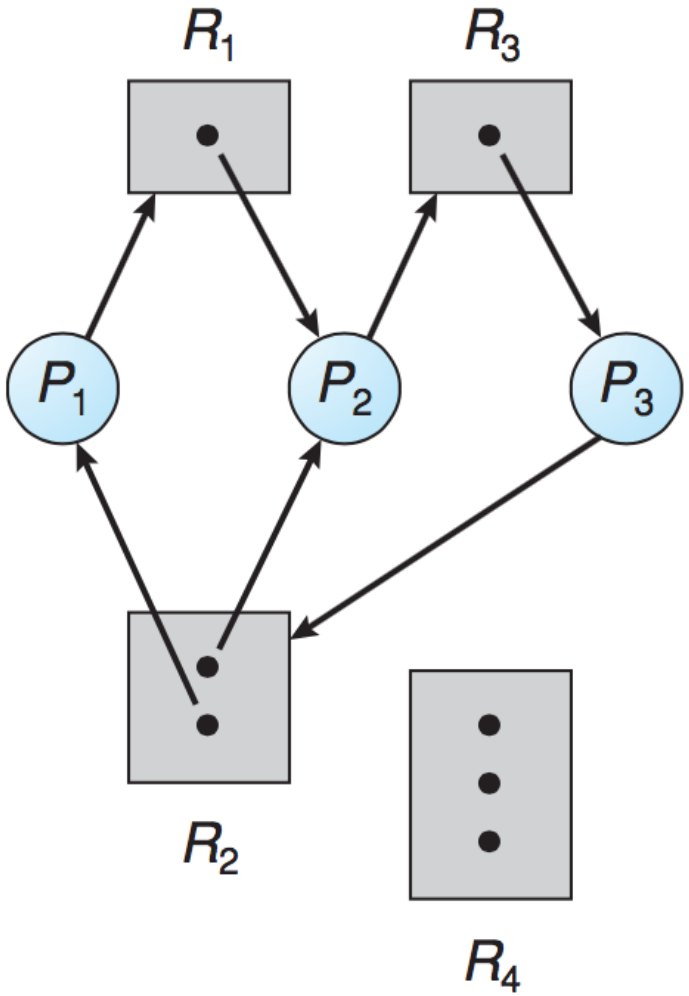
If the first three conditions are true, deadlock is possible, but deadlock will only happen if the fourth condition is fulfilled.

Resource Allocation Graph



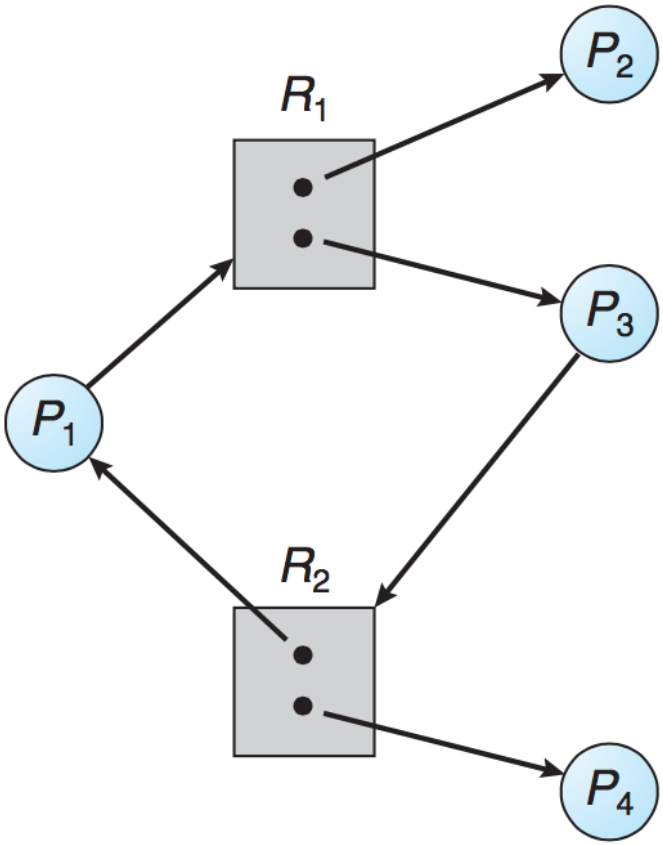
Resource Allocation Graph

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



Resource Allocation Graph

But the presence of a cycle is not necessarily certain that there is a deadlock.



Dealing with Deadlock

There are four basic approaches to dealing with deadlock.

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

Ignoring Deadlock

This option is certainly convenient for operating system designers!

They simply pretend that deadlock can never happen.

Or if it does happen, it is the application developers' fault.

It doesn't seem right, but… Microsoft Windows takes this approach.

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.

But can you remove the need for it in your program?

Knocking Down: Mutual Exclusion

Mutual Exclusion cannot, generally speaking, be disallowed.

The purpose of mutual exclusion is: prevent errors like inconsistent state/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.

Knocking Down 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.

This does not mean that things can be requested only one at a time.

One plausible solution: request all resources at the beginning of the program.

If the program needs R_1, R_2, and R_3 at some point, all three must be requested right at the beginning and held throughout the program.

Knocking Down Hold-and-Wait

A process has to know in advance all of the resources that it will need.

Remember that a file is a resource.

A simple text editor can be used by the user to open an arbitrary file.

How do we know in advance which will be requested?

Knocking Down Hold-and-Wait

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.

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).

Knocking Down Hold-and-Wait

What if a process must release all its currently-held resources before it can get any new ones?

The process has R_1 and R_2 and wants to get R_3.
First release R_1 and R_2 before it can request all three.

A resource that cannot be easily released is memory.
Released memory may be collected and reassigned by the operating system.

Therefore we cannot release all resources.

We cannot rule out deadlock; we can only make it less likely to occur.

Two-Phase Locking

Another idea that might work is two-phase locking.

A process attempts to lock a group of resources at once.

If it does not get everything it needs, it releases the locks it got 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.

Two-Phase Locking

Two phase locking is not applicable to our current model for semaphores.

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.

There are systems that have nonblocking requests & mutual exclusion.

Then the program is responsible for checking if any of the requests returned `false` and releasing any resources where the request returned `true`.

Try This

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.

That is the only way to know if the lock was acquired.

Have You Tried Trying?

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 );
```

Knocking Down No Preemption

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 P_2 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.

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.

Allowing Preemption

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 isn't really something we can do as program designers.

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.