**6.6. Synchronization and Deadlock**

The Dining Philosophers Problem illustrates how careless use of synchronization primitives can lead to *Deadlock*. Specifically, recall that the following four conditions are necessary for deadlock to occur:

* Mutual exclusion: Only one thread can hold a single resource.
* No preemption: Once a thread acquires a resource, no other thread can steal it.
* Hold and wait: Threads can hold one resource while trying to acquire another.
* Circular wait: Multiple threads are simultaneously waiting on resources held by each other.

6.6.1. The Dining Philosophers Problem

The Dining Philosophers Problem illustrates deadlock by envisioning a table of philosophers sitting at a circular table under the following conditions:

* Each philosopher rotates through three states: thinking, hungry, and eating.
* At the center of the table is a bowl of spaghetti. When a philosopher gets hungry, they serve themselves from the bowl and eat.
* Between each pair of philosophers is a single serving fork. Each philosopher can serve themselves by grabbing the fork to their left and the fork to their right, then scoop food onto their plate.
* Once a philosopher grabs one fork, they hold it until they can grab the other and eat.

The following image (source: Wikipedia [**[1]**](https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/cs361/html/ClassicalDeadlock.html#id2)) illustrates the circular relationship between the philosophers and their forks.



The Dining Philosophers Problem arises any time that concurrent threads need to access more than one mutually exclusive resource simultaneously. For instance, consider a banking system that is transferring money between two accounts. To ensure the transfer is correct, the system locks both accounts before initiating the exchange. Once the exchange has completed, both account locks are released. Failure to handle this case carefully can lead to deadlock of those resources.

6.6.2. A failed solution attempt

As an initial attempt to solve this problem, consider the following basic code sample that illustrates the algorithm.

**void** \*

philosopher (**void** \* \_args)

{

*/\* Each of the N philosophers is identified by index i \*/*

*/\* The N forks are numbered from 0 to N-1 \*/*

*/\* Cast the args to a usable struct type \*/*

**struct** args \*args = (**struct** args \*) \_args;

**while** (1)

{

THINK();

*/\* Grab the left fork \*/*

pthread\_mutex\_lock (args->fork[i]);

*/\* Grab the right fork \*/*

pthread\_mutex\_lock (args->forks[(i+1) % N]);

EAT();

*/\* Release the forks \*/*

pthread\_mutex\_unlock (args->fork[i]);

pthread\_mutex\_unlock (args->forks[(i+1) % N]);

}

}

In abstract terms, the algorithm shown here starts off with a non-critical section of code (THINK()) that does not require mutual exclusion. Then, to enter the critical section (EAT()), the thread must acquire two locks, which are both released after the critical section. That is, this problem is similar to a basic critical section problem, except more than one lock is required.

To illustrate the deadlock that can arise, consider an instance where there are 3 threads and 3 locks (*i.e.,*3 philosophers and 3 forks). The following sequence of events leads to deadlock:

* Thread 0 acquires lock 0
* Thread 1 acquires lock 1
* Thread 2 acquires lock 2
* Thread 0 tries to acquire lock 1 (blocked by thread 1)
* Thread 1 tries to acquire lock 2 (blocked by thread 2)
* Thread 2 tries to acquire lock 0 (blocked by thread 0)

At this point, thread 0 is blocked by thread 1, thread 1 is blocked by thread 2, and thread 2 is blocked by thread 0. Hence, we have a circular wait and the system is deadlocked.

6.6.3. Possible solutions

One solution to the Dining Philosophers Problem is to use a second semaphore that controls access to the table. Specifically, we require that there must always be at least one open seat and philosophers have to do their thinking away from the table. The implementation of this approach is straightforward. Initialize a sit\_down semaphore to N-1 for N philosophers. That is, if there are 5 philosophers, only 4 can sit at the table at a time. The new loop would look like the following code.

THINK();

*/\* Try to sit down at the table \*/*

sem\_wait (args->sit\_down);

*/\* Grab the left fork \*/*

pthread\_mutex\_lock (args->fork[i]);

*/\* Grab the right fork \*/*

pthread\_mutex\_lock (args->forks[(i+1) % N]);

EAT();

*/\* Release the forks \*/*

pthread\_mutex\_unlock (args->fork[i]);

pthread\_mutex\_unlock (args->forks[(i+1) % N]);

*/\* Get up from the table \*/*

sem\_post (args->sit\_down);

Returning to the scenario above (with 3 philosophers), note the new sequence of events would be as follows.

* Thread 0 waits on sit\_down
* Thread 0 acquires lock 0
* Thread 1 waits on sit\_down
* Thread 1 acquires lock 1
* Thread 2 waits on sit\_down; thread 2 is now blocked
* Thread 0 tries to acquire lock 1 (blocked by thread 1)
* Thread 1 acquires lock 2

By limiting the number of concurrent accesses to the table, we can guarantee that there will always be enough forks available and the circular wait will be prevented.

While the previous approach works, it has an undesirable characteristic. To prevent deadlock of one resource (the forks), we are forcing another resource (the seats) to be underutilized. In some settings, this choice might be a poor trade-off. In the case of the philosophers, perhaps that last philosopher is really tired and just wants to sit down while they are waiting; they promise that they will note grab a fork until somebody leaves. Back in technical terms, the approach outlined above would prevent a thread from making initial progress on the resource protected by the semaphore. This delay may cause unfortunate side effects elsewhere.

An alternative solution to the Dining Philosopher Problem is to impose a linear ordering on the forks. That is, fork 0 must be acquired before fork 1, fork 1 must be acquired before fork 2, and so on. The code for this solution is as follows.

THINK();

left = i;

right = (i + 1) % N;

*/\* The last philosopher needs to change the order of the forks \*/*

**if** (left > right)

{

left = right;

right = left + 1;

}

*/\* Grab the left fork \*/*

pthread\_mutex\_lock (args->fork[left]);

*/\* Grab the right fork \*/*

pthread\_mutex\_lock (args->forks[right]);

EAT();

*/\* Release the forks \*/*

pthread\_mutex\_unlock (args->fork[left]);

pthread\_mutex\_unlock (args->forks[right]);

The key to this observation is that one philosopher must end up grabbing their right fork before their left. That is, if there are 10 philosophers (it's a really big table), then philosopher 9 would initially calculate left = 9 and right = 0 (since (9 + 1) % 10 is 0). Since left would be bigger than right, this philosopher (and only this philosopher) would switch their order. Specifically, they would try to grab fork 0 before fork 9.

Returning to the example of 3 threads and 3 locks, the order of events would be as follows.

* Thread 0 acquires lock 0
* Thread 1 acquires lock 1
* Thread 2 tries to acquire lock 0 (blocked by thread 0)
* Thread 0 tries to acquire lock 1 (blocked by thread 1)
* Thread 1 acquires lock 2

At this point, thread 1 can complete its work. When it releases the two locks, thread 0 will be able to acquire lock 1 and continue. Eventually, thread 0 would also release both locks and thread 2 could continue.