If the necessary synchronization requires acquiring more than one lock, there are some special considerations that need to be taken into account.

For example, the code below implements the transfer of funds from one bank account to another.

The code assumes there is a separate semaphore lock for each account and since it needs to adjust the balance of two accounts, it acquires the lock for each account.

Consider what happens if two customers try simultaneous transfers between their two accounts.

The top customer will try to acquire the locks for accounts 6005 and 6004.

The bottom customer tries to acquire the same locks, but in the opposite order.

Once a customer has acquired both locks, the transfer code will complete, releasing the locks.

But what happens if the top customer acquires his first lock (for account 6005) and the bottom customer simultaneously acquires his first lock (for account 6004).

So far, so good, but now each customer will be not be successful in acquiring their second lock, since those locks are already held by the other customer!

This situation is called a "deadlock" or "deadly embrace" because there is no way execution for either process will resume.

Both will wait indefinitely to acquire a lock that will never be available.

Obviously, synchronization involving multiple resources requires a bit more thought.

The problem of deadlock is elegantly illustrated by the Dining Philosophers problem.

Here there are, say, 5 philosophers waiting to eat.

Each requires two chopsticks in order to proceed, and there are 5 chopsticks on the table.

The philosophers follow a simple algorithm.

First they pick up the chopstick on their left, then the chopstick on their right.

When they have both chopsticks they eat until they're done, at which point they return both chopsticks to the table, perhaps enabling one of their neighbors to pick them up and begin eating.

Again, we see the basic setup of needing two (or more) resources before the task can complete.

Hopefully you can see the problem that may arise If all philosophers pick up the chopstick on their left, then all the chopsticks have been acquired, and none of the philosophers will be able to acquire their second chopstick and eat.

Another deadlock!

Here are the conditions required for a deadlock: 1.

Mutual exclusion, where a particular resource can only be acquired by one process at a time.

2.

Hold-and-wait, where a process holds allocated resources while waiting to acquire the next resource.

3.

No preemption, where a resource cannot be removed from the process which acquired it.

Resources are only released after the process has completed its transaction.

4.

Circular wait, where resources needed by one process are held by another, and vice versa.

How can we solve the problem of deadlocks when acquiring multiple resources?

Either we avoid the problem to begin with, or we detect that deadlock has occurred and implement a recovery strategy.

Both techniques are used in practice.

In the Dining Philosophers problem, deadlock can be avoided with a small modification to the algorithm.

We start by assigning a unique number to each chopstick to establish a global ordering of all the resources, then rewrite the code to acquire resources using the global ordering to determine which resource to acquire first, which second, and so on.

With the chopsticks numbered, the philosophers pick up the lowest-numbered chopstick from either their left or right.

Then they pick up the other, higher-numbered chopstick, eat, and then return the chopsticks to the table.

How does this avoid deadlock?

Deadlock happens when all the chopsticks have been picked up but no philosopher can eat.

If all the chopsticks have been been picked up, that means some philosopher has picked up the highest-numbered chopstick and so must have earlier picked up the lower-numbered chopstick on his other side.

So that philosopher can eat then return both chopsticks to the table, breaking the hold-and-wait cycle.

So if all the processes in the system can agree upon a global ordering for the resources they require, then acquire them in order, there will be no possibility of a deadlock caused by a hold-and-wait cycle.

A global ordering is easy to arrange in our banking code for the transfer transaction.

We'll modify the code to first acquire the lock for the lower-numbered account, then acquire the lock for the higher-numbered account.

Now, both customers will first try to acquire the lock for the 6004 account.

The customer that succeeds then can acquire the lock for the 6005 account and complete the transaction.

The key to deadlock avoidance was that customers contented for the lock for the *first* resource they both needed.

Acquiring that lock ensured they would be able to acquire the remainder of the shared resources without fear that they would already be allocated to another process in a way that could cause a hold-and-wait cycle.

Establishing and using a global order for shared resources is possible when we can modify all processes to cooperate.

Avoiding deadlock without changing the processes is a harder problem.

For example, at the operating system level, it would be possible to modify the WAIT SVC to detect circular wait and terminate one of the WAITing processes, releasing its resources and breaking the deadlock.

The other strategy we mentioned was detection and recovery.

Database systems detect when there's been an external access to the shared data used by a particular transaction, which causes the database to abort the transaction.

When issuing a transaction to a database, the programmer specifies what should happen if the transaction is aborted, e.g., she can specify that the transaction be retried.

The database remembers all the changes to shared data that happen during a transaction and only changes the master copy of the shared data when it is sure that the transaction will not be aborted, at which point the changes are committed to the database.

In summary, we saw that organizing an application as communicating processes is often a convenient way to go.

We used semaphores to synchronize the execution of the different processes, providing guarantees that certain precedence constraints would be met, even between statements in different processes.

We also introduced the notion of critical code sections and mutual exclusion constraints that guaranteed that a code sequence would be executed without interruption by another process.

We saw that semaphores could also be used to implement those mutual exclusion constraints.

Finally we discussed the problem of deadlock that can occur when multiple processes must acquire multiple shared resources, and we proposed several solutions based on a global ordering of resources or the ability to restart a transaction.

Synchronization primitives play a key role in the world of "big data" where there are vast amounts of shared data, or when trying to coordinate the execution of thousands of processes in the cloud.

Understanding synchronization issues and their solutions is a key skill when writing most modern applications.