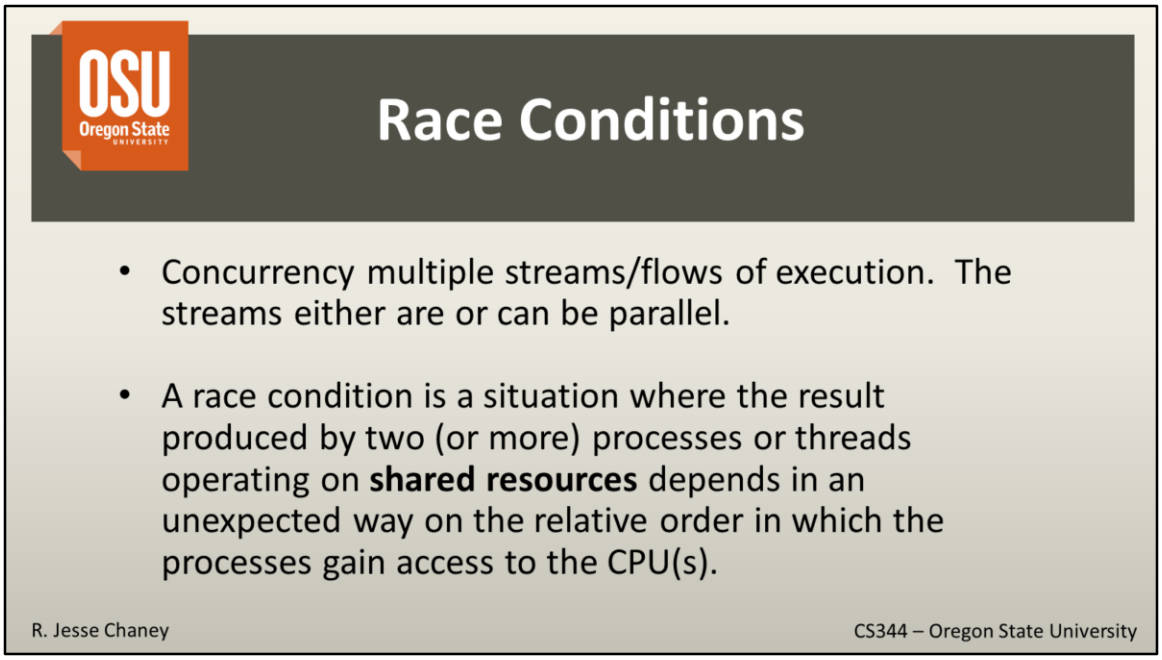
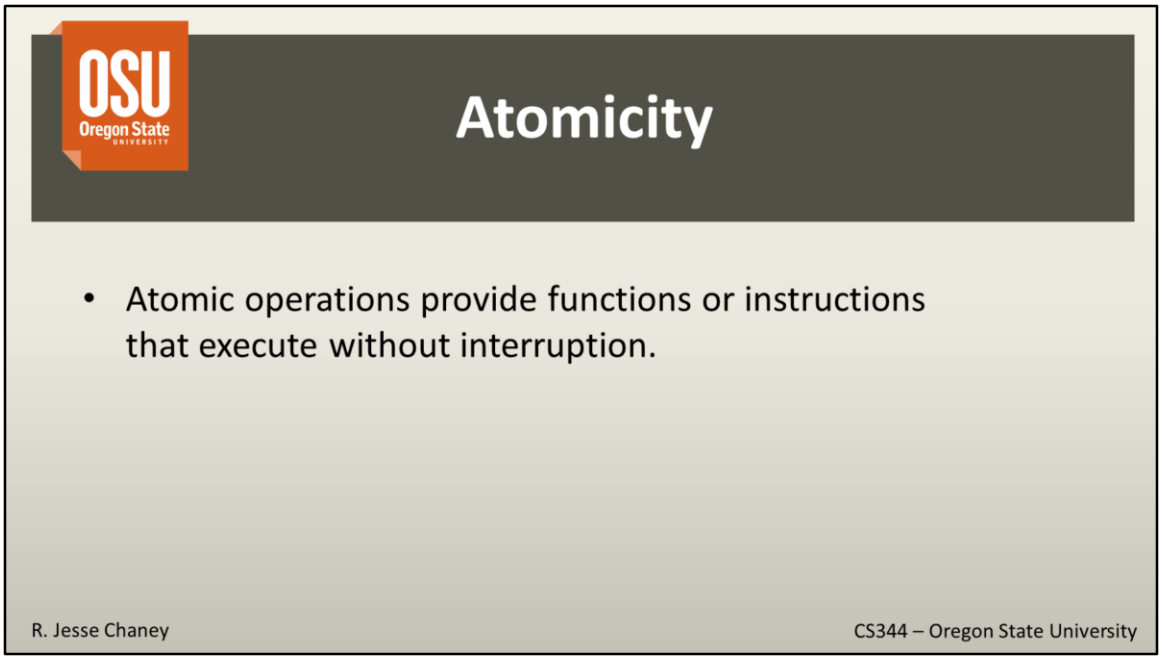


One of the terms that will come up often is concurrency. It simple means that there are related streams (or flows) of execution that do or could run at the same time (in parallel). If there is more than 1 CPU, the streams of execution may run in parallel. The streams of execution could be on different systems, but still access a shared resource.

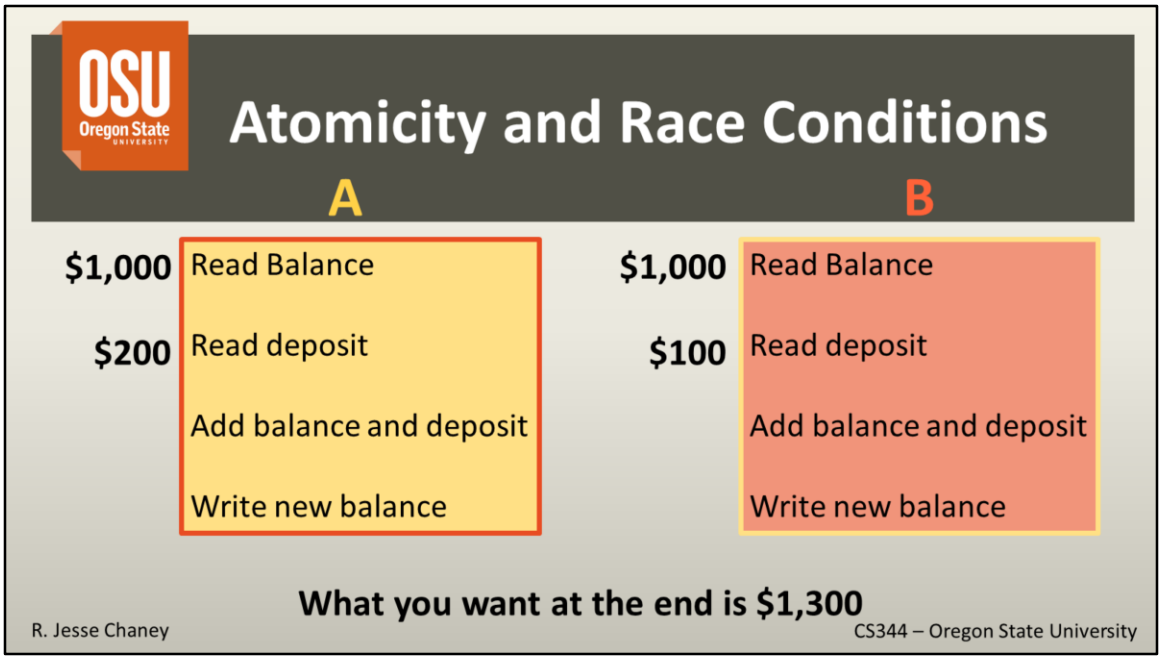
Processes and threads are concurrent streams of execution.

A shared resource can be just about anything: a file, a row in a database, a device, …



Atomic operations are an important way to manage access to shared resources.

Atomic operations are used in many modern operating systems and parallel processing systems.



A shared resource, your account balance.

If we serialize everything, it all goes well. Process A reads the balance. Reads the deposit. Add the deposit to the balance. And then writes the updated balance into the store, yielding a new balance of $1,200. Then process B comes along and reads that updated balance ($1,200). Reads the next deposit. Adds the deposit to the balance. And, finally writes the new balance back into the database. You wind up with $1,300. That is the result you want to see.

If there are 2 deposits going on at the same time, we can get into some strange situations. In our example, there are just 2 processes, but you can imagine that if there are a large number of processes all trying to deposit money into your account that you could miss a few transitions. In banking, and many other industries, you don't like to miss transactions.

Imagine this scenario. Processes A and B are running at the some time, in different cities, maybe even different continents.
If we have process A read the account balance of $1000 then have process B read the account balance of $1000. Then process A adds the deposit of $200 to its copy of the

balance. And process B add the deposit of $100 to its copy of the balance. Now process A writes its value of the balance into the account. Right after that, process B writes its copy of the account balance into the account. What you come out with is an account balance of $1,100. Not what you want! If it happens the other way around, process B writes the new balance first followed by process A, you wind up with a balance of $1,200, which is better than $1,100, but still not the $1,300 that you should have in the account.

The order of operations determines the end result. We don't like that.

We get a race condition. There is a shared resource. The order of access to the shared resource can determine the outcome of the sequence of events. That is a race condition. We want to create the conditions where we always know the outcome and the outcome is what we want. This is an example of a condition where the process steps have to run in a serialized fashion. They both try to operate on the account balance, in 2 places.

One way to create safety around sequences of operation like this is to create locks on the access to the shared resources. We will get into some locks. When we are doing multi-processing and multi-threading that allow us to lock shared resources.

Another way to do this is to make the operations Atomic. We atomically update the balance in process A and then atomically update the balance in process B. We make the sequence of events a single atomic operation. Once process A begins, it cannot be interrupted by another process that might alter the outcome. Process B must wait until the atomic operation is completed by A before it can begin. At that point, B will see the updated balance from the completion of process A. Or, it could happen the other way around, process B precedes Process A. The outcome is the same though, an account balance of $1,300.

Another fundamental feature of atomic operations is that they either completely succeed or they completely fail. In our case, if the atomic operation of process A were proceed process B and process A's atomic operation were to fail, then the account balance would remain at $1,000 **AND** the deposit of $200 would still be in the queue. The transaction would completely rollback. We don't partially read the balance. We don't partially read the deposit. And, we don't partially update the balance. Being atomic, it is all or nothing.

Some additional examples of race conditions are given in your book, section 5.1.

Reverse a File