Two threads are **concurrent** if their flows overlap in time

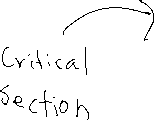
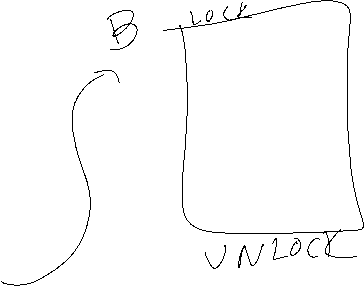
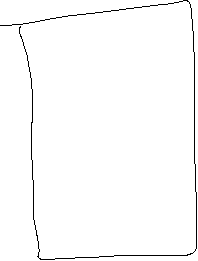
Threads that are not concurrent are called **sequential**

**Single core**-simulate **parallelism** (running at the same time) by time slicing

**Multi-core**-true parallelism possible

**Synchronization**

We run into trouble when two processes that are sharing something are concurrent.



Race Condition: see slides

Need to identify what is shared: address space, descriptors, code, data, heap

Atomic: an operation is atomic if it runs completely OR does not run at all

Examples of what’s atomic:

LOADS and STORES are probably atomic

Many other instructions are not

Mutual Exlcusion: only one thread executes a particular hunk of code at a time

Critical Section: the hunk of code that only one thread should ever be in at a time

Remember that there could be many critical sections

Synchronization: just controlling the timing of access to shared resources

See slides

What is going to be shared among threads:

Global variables (data section)

Malloc’ed memory (heap)

Static local variables (data section)

Stores variable in data section where globals go, making it visible everywhere;

Pointers to another thread’s stack space

Files

Other system resources

Before accessing shared state, we need to lock something, and then at the end, unlock it

How do we build locks?

There are many ways

What do we want?

To be the only thread in the critical session

To run it automatically

If you are in the middle of a critical section, no other thread gets to run where they can manipulate the shared state/the critical section.

How do we lose control of the processor?

Interrupt

We do a system call

So basically, we want to lock the system calls. Can we?

We cannot let users turn off the interrupts because this could cause a lot of problems (break the computer)

The critical section could be really long, so we wouldn’t want the interrupt to be turned off

With locks, you pay for the performance with correctness

Busy waiting: while constantly checking it works, constantly asking “are we there yet” is not only annoying, but we are actively consuming resources (CPU, taking up bandwidth on memory disk)

If we have a ton of cpus and system resources, then sure, it can be okay. But for the majority of cases, this inefficiency is concerning.

Semaphores- particular algorithm developed by Djikstra

With a semaphore, we got that its state is an int

When we create it, we give it an int value

Two allowable operations: P() and V() for increment and decrement()

P(s)  
▶ if nonzero:  
▶ decrement  
▶ return immediately  
▶ (note that test and decrement are atomic)  
▶ else (s is 0)  
▶ put thread to sleep until s >0 and thread restarted by a V()

▶ upon restart P()  
▶ decrements s  
▶ returns control to calling thread

V(s)  
▶ increment s (occurs atomically)  
▶ if any threads blocked in a P() on S  
▶ restart exactly one blocked thread  
▶ previously blocked thread completes its P() and decrements s .