Critical sections

Example:

If we run the Pthreads program with two threads and n is relatively small, we find that the results of the Pthreads program are in agreement with the serial sum program. However, as n gets larger, we start getting some peculiar results. For example, with a dual-core processor we get the following results:

	n			
	10^5	10^6	10^7	10^8
π	3.14159	3.141593	3.1415927	3.14159265
1 Thread	3.14158	3.141592	3.1415926	3.14159264
2 Threads	3.14158	3.141480	3.1413692	3.14164686

Notice that as we increase n, the estimate with one thread gets better and better. In fact, with each factor of 10 increase in n, we get another correct digit. With $n = 10^5$, the result as computed by a single thread has five correct digits. With $n = 10^6$, it has six correct digits, and so on. The result computed by two threads agrees with the result computed by one thread when $n = 10^5$. However, for larger values of n, the result computed by two threads actually gets worse. In fact, if we ran the program several times with two threads and the same value of n, we would see that the result computed by two threads *changes* from run to run. The answer to our original question must clearly be, "Yes, it matters if multiple threads try to update a single shared variable."

Let's recall why this is the case. Remember that the addition of two values is typically *not* a single machine instruction. For example, although we can add the contents of a memory location `y` to a memory location `x` with a single C statement,

```
x = x + y;
```

what the machine does is typically more complicated. The current values stored in `x` and `y` will, in general, be stored in the computer's main memory, which has no circuitry for carrying out arithmetic operations. Before the addition can be carried out, the values stored in `x` and `y` may therefore have to be transferred from main memory to registers in the CPU. Once the values are in registers, the addition can be carried out. After the addition is completed, the result may have to be transferred from a register back to memory.

```
y = Compute(my_rank);
x = x + y;
```

Let's also suppose that thread 0 computes `y = 1` and thread 1 computes `y = 2`. The "correct" result should then be `x = 3`. Here's one possible scenario:

Time	Thread 0	Thread 1
1	Started by main thread	
2	Call `Compute()`	Started by main thread
3	Assign `y = 1`	Call `Compute()`
4	Put `x=0` and `y=1` into registers	Assign `y = 2`
5	Add 0 and 1	Put `x=0` and `y=2` into registers
6	Store 1 in memory location `x`	Add 0 and 2
7		Store 2 in memory location `x`

So we see that if thread 1 copies `x` from memory to a register *before* thread 0 stores its result, the computation carried out by thread 0 will be *overwritten* by thread 1. The problem could be reversed: if thread 1 *races* ahead of thread 0, then its result may be overwritten by thread 0. In fact, unless one of the threads stores its result *before* the other thread starts reading `x` from memory, the "winner's" result will be overwritten by the "loser."

Notes:

- In this pi program example, when we only have one thread running the program we see that as we increase n, the closer we get to pi.
- But when we run 2 threads with the program we see that as we increase n, we get worse results.
- Why is that? This is because when we try and modify shared variables b/w threads we get race conditions which can severely mess up computations and results

Note:

- As you can see Thread 0 is known as the "winner" because it finishes faster than thread 1 "loser" but what happens in a race condition is that the "loser" ends up overwriting what the "winner" thread had which can effect the outcome of the program in an unwanted way

- In this example, x = x + y is the code with a shared resource that we identify as x.
    - I think its because x is being used to add with y and then stored which if x is overwritten by another thread it

can effect the addition.

- This code is known as a **critical section**, that is, it's a block of code that updates a shared resource that can only be updated by one thread at a time.

Race conditions and buggy critical sections are hard to identify and debug:

These types of issues are particularly difficult to debug, because the outcome is non-deterministic. It is entirely possible that the error shown above occurs less than 1% of the time and could be influenced by external factors, including the hardware, operating system, or process scheduling algorithm. Even worse, attaching a debugger or adding `printf` statements to the code may change the relative timing of the threads and seemingly "correct" the issue temporarily. Such bugs that disappear when inspected are known as *Heisenbugs* (the act of observing the system alters its state).