Operating Systems!

Concurrency:
Race Conditions

Prof. Travis Peters

Montana State University

CS 460 - Operating Systems

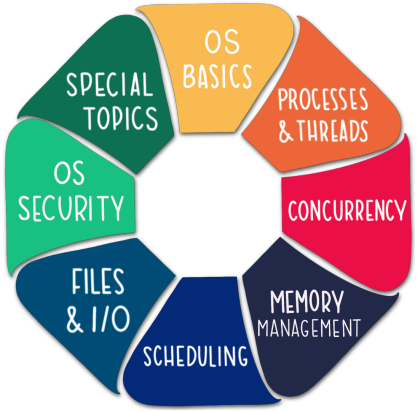
Fall 2020

https://www.cs.montana.edu/cs460

Some diagrams and notes used in this slide deck have been adapted from Sean Smith's OS courses @ Dartmouth. Thanks, Sean!

Today

Travis has already helped a few "Free Agents" form team members! Great to have a group to discuss things (e.g., Accountability, PA2, Studying, Yalnix CP1)



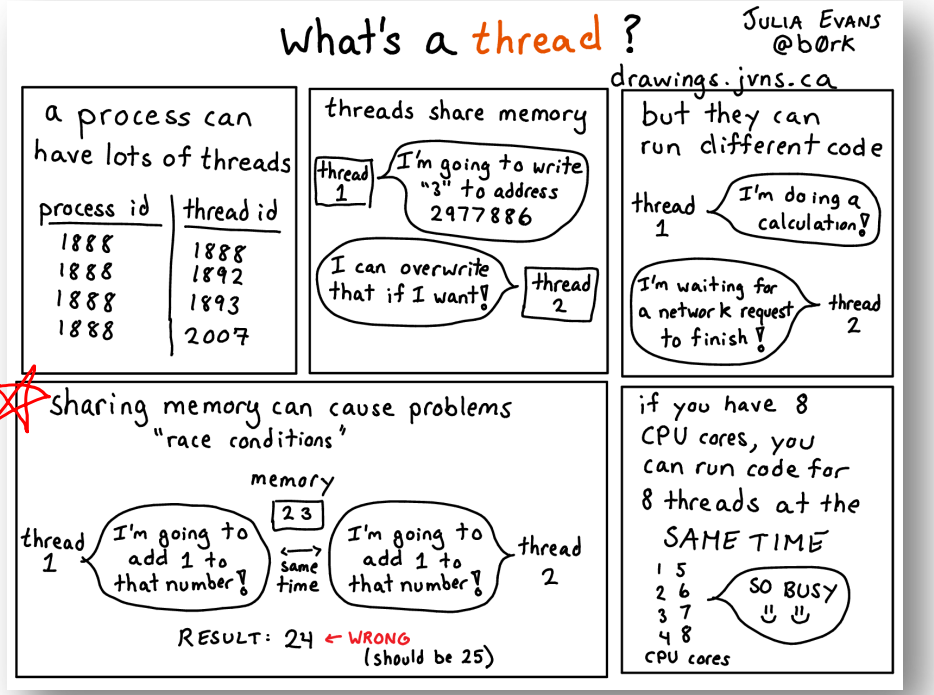
- Announcements

Read over the exam coversheet!

  - *Heads up…. Exam 1:* ***Friday [10/02/2020] – due @ 11:59 PM (MST)***
  - *Heads up…. PA2 due* ***Sunday [10/04/2020] @ 11:59 PM (MST)***
  - *Heads up*…. Yalnix! Form teams *ASAP!* ☺

Check out the PA2 starter files!

- Learning Objectives
  - *Threads & Interleaving*
  - *What are "Race Conditions"?*
  - *Examples*
  - *Towards a Solution*



Last Time...

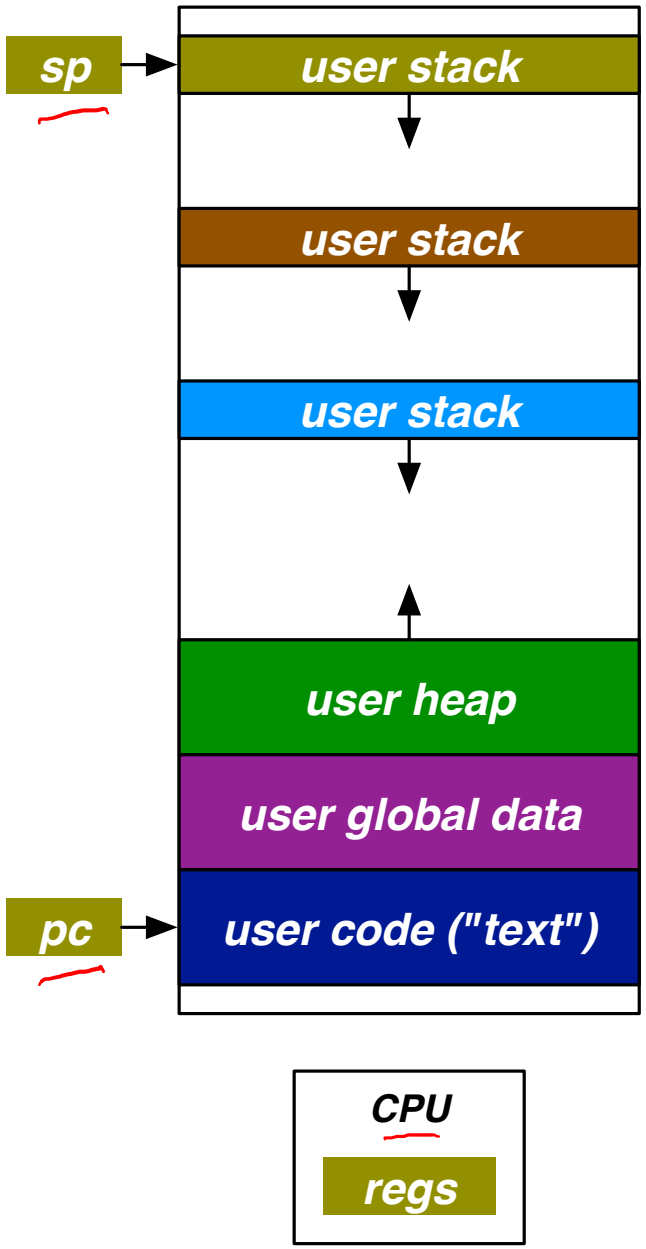
- Threads "below the surface"
  - userlevel threads vs. kernellevel threads
- Threads "above the surface" (**pthreads**)
  - `pthread_t mythread;`
  - `pthread_create(...);`
  - `pthread_join(...);`
  - `pthread_cancel(...);`
  - `...`

similar (not the same)
fork
wait
exit/abort/kill

thread-related demos:

- *threads0.c*
- *threads1.c*
- *threads2.c*
- *thread_spy.c*
- *thread_atm.c*
- *ipt.c*

IPC



Concurrency – *what* is it? *why* is it?

- **Concurrency** is all about communicating & sharing whilst **interleaving** and **overlapping** execution.
  - **Note:** We'll mostly focus on **uniprocessor systems**
  - **Recall:** A **thread** is a single (separately schedulable) unit of execution

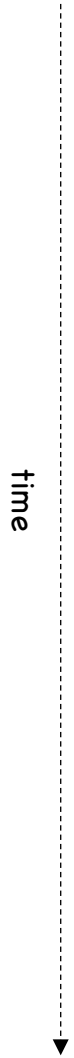
- Examples
  - Servers ➔ managing multiple connections
  - Programs w/ UIs ➔ responsiveness
  - Network-/Disk-Bound Programs ➔ hide network/disk latency

Threads & Interleaving

➔ See ipt.c

A Simple Threaded Program

Thread A Thread B

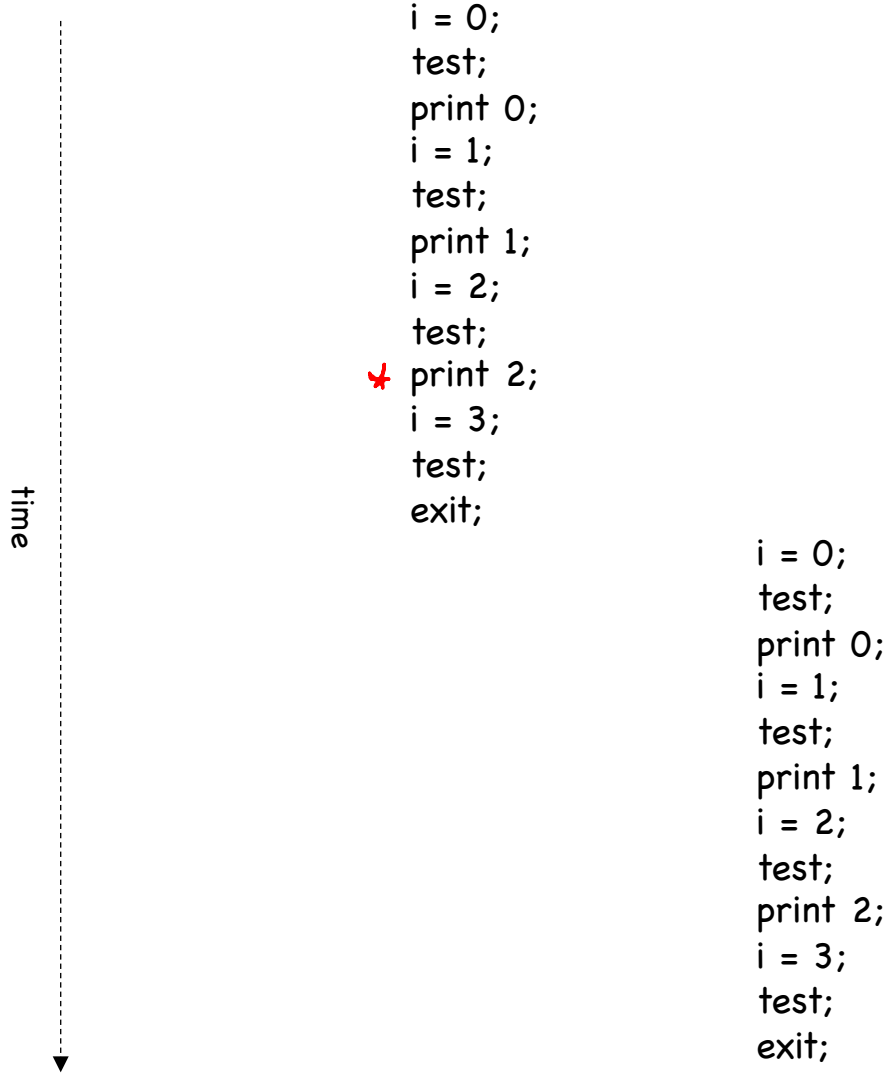


Consider code:

```
int i; //global variable

...
func {
    for (i = 0; i < 3; i++) {
        printf("%d\n", i);
    }
}
```

A Simple Threaded Program

Maybe they will be well-behaved...



Consider code:

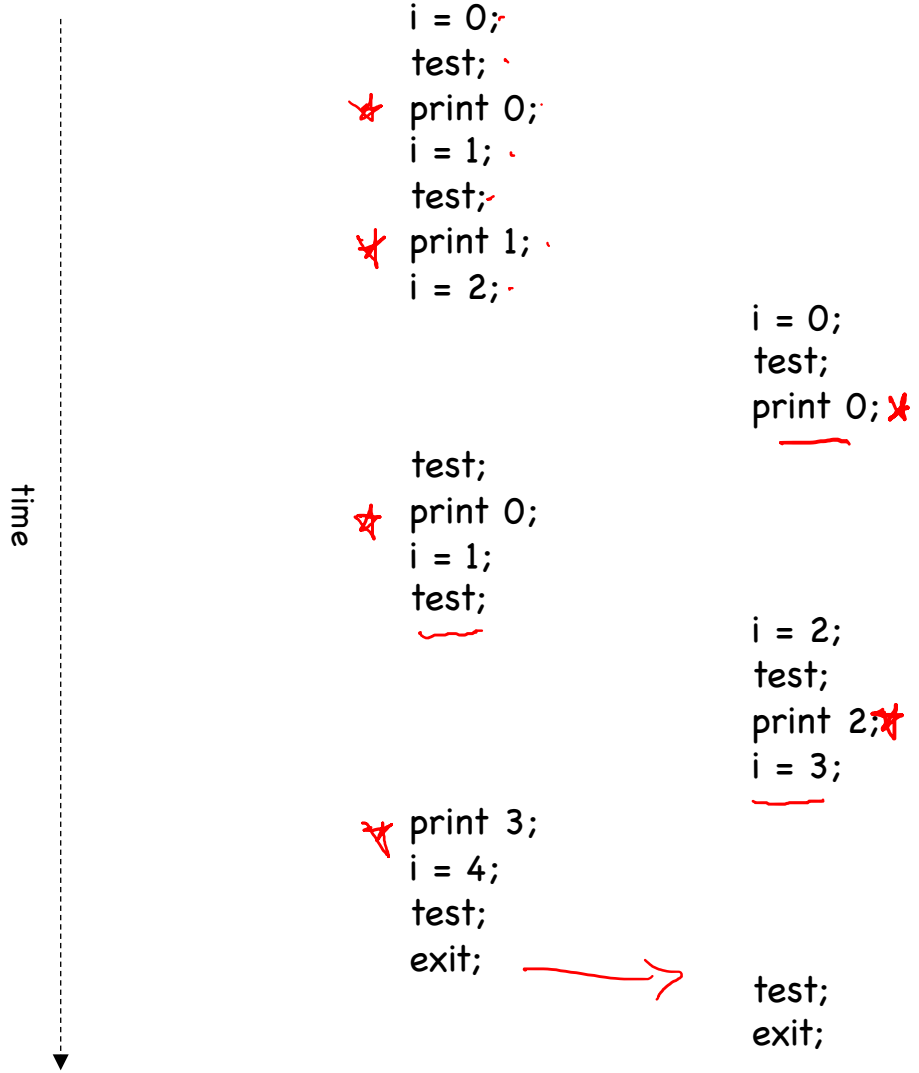
```
int i; //global variable

…

for (i = 0; i < 3; i++) {
    printf("%d\n", i);
}
```

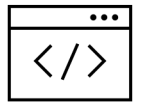
A Simple Threaded Program

...but maybe they won't be well-behaved!

Consider code:

```
int i; //global variable

...

for (i = 0; i < 3; i++) {
    printf("%d\n", i);
}
```





printf
int

Why Does This Happen?

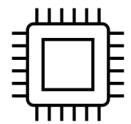
What happens when a program executes?

compiler

- High-level code
  ➔ sequence of assembly instructions

- Execute instructions *one instruction at a time*

- Can be *interrupted / pre-empted at any time!*



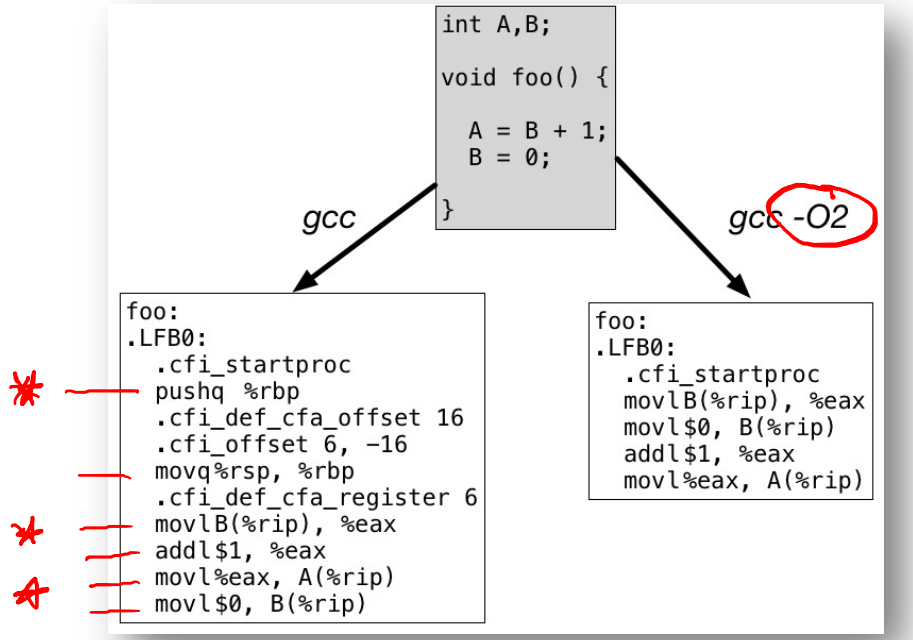
Why Does This Happen?

What happens when a program executes?

- High-level code
  ➔ sequence of assembly instructions

- Execute instructions *one instruction at a time*

- Can be *interrupted / pre-empted at any time!*



https://preshing.com/20120625/memory-ordering-at-compile-time/

Also: compiler re-ordering...
*...lots of non-intuitive things, all in the name of **better performance**!*

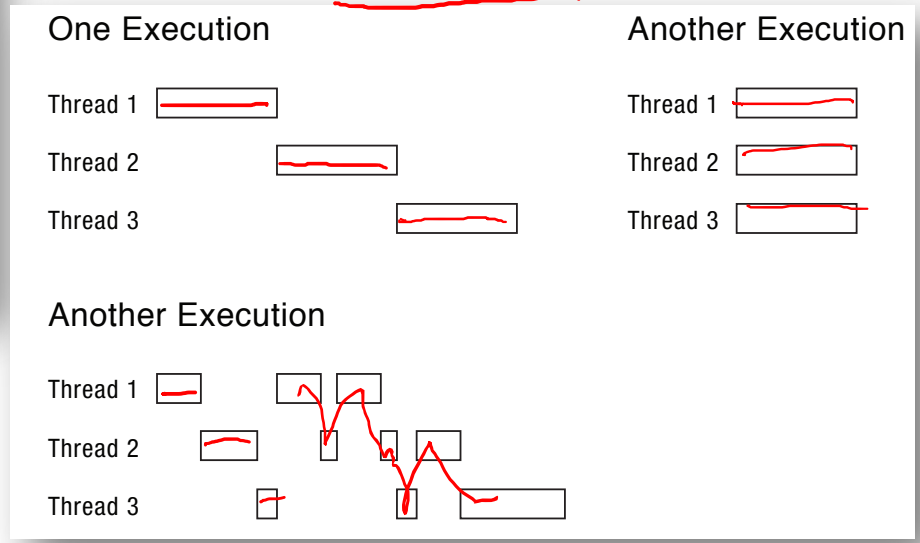
Programmer vs. Processor View

programmer view:

Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
.	.	.	.
x = x + 1;	x = x + 1;	x = x + 1;	x = x + 1;
y = y + x;	y = y + x;	.....	y = y + x;
z = x + 5y;	z = x + 5y;	Thread is suspended.	.....
.	.	Other thread(s) run.	Thread is suspended.
.	.	Thread is resumed.	Other thread(s) run.
.	.	.....	Thread is resumed.
		y = y + x;	.....
		z = x + 5y;	z = x + 5y;

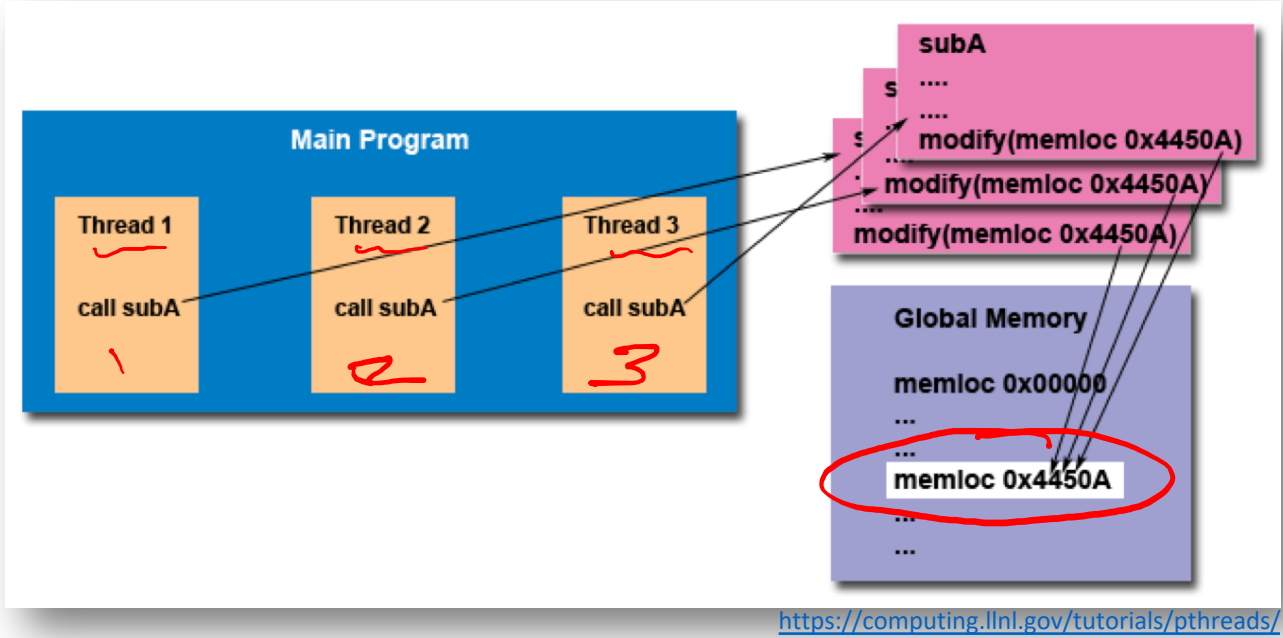
Programmer/program must anticipate all of these possible executions!

processor view:



Race Conditions

Race Conditions



https://computing.llnl.gov/tutorials/pthreads/

- What does it mean for a computation to be "correct"?

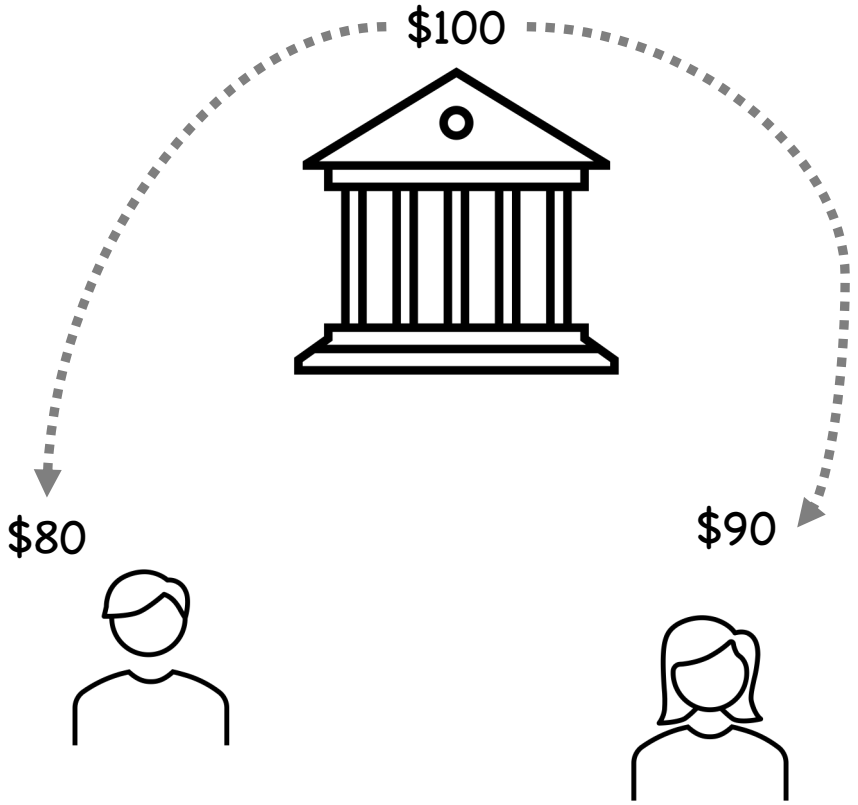
- **Race Condition:** when not all observable outcomes are correct.
  ➔ *Whether the program works depends on a "race" between concurrent threads or processes.*

*Put another way: when **shared data** is **corrupted** due to **bad interleavings** / thread scheduling*

Race Conditions

Banking Example

- Alice and Bob share a bank account w/ $100.
- Both want to make withdraws...
  - Alice asks for $90
  - Bob asks for $80



Q: What are correct outcomes?

Alice get $90
Bob gets $0 "

Q: What might *actually* happen?

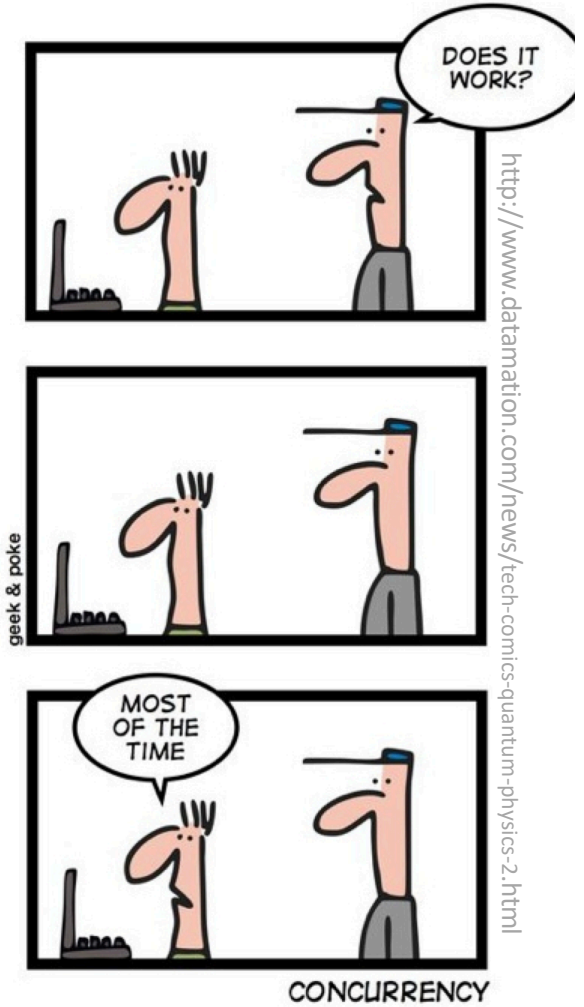
" "

➔ See thread_atm.c

Debugging Race Conditions

Race conditions are tricky...

- It may be that only some interleavings are problematic
  - Code works fine (most of the time...)
  - try to debug... can't recreate the problem...
- External factors may change interleavings
  - different systems?
  - using a debugger? ("heisenbugs")



Practical Manifestations

When and where should you look for race conditions in your code? Here are some examples...

- When you have **shared data**
- When you have a data structure with multiple fields
  ➔changes take several steps
- When you test something, *then* act on the result of the test
  ➔ **TOCTOU** ("Time of Check vs. Time of Use")
- When the action of one thread requires a prior action by another thread
- When you want to wait until two different things are both true

Towards a Solution

Critical Section

How could we
"guard"
a critical section?

```
while(flag) {}
flag = TRUE;
```

Critical Section

```
flag = FALSE;
```

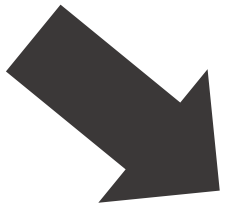
```
while(flag) {}
flag = TRUE;
Critical Section
flag = FALSE;
```

```
while(flag) {}
flag = TRUE;
Critical Section
flag = FALSE;
```

```
while(flag) {}
flag = TRUE;

Critical Section

flag = FALSE;
```

SW → TOCTOU

Atomic "Test and Set" Instruction

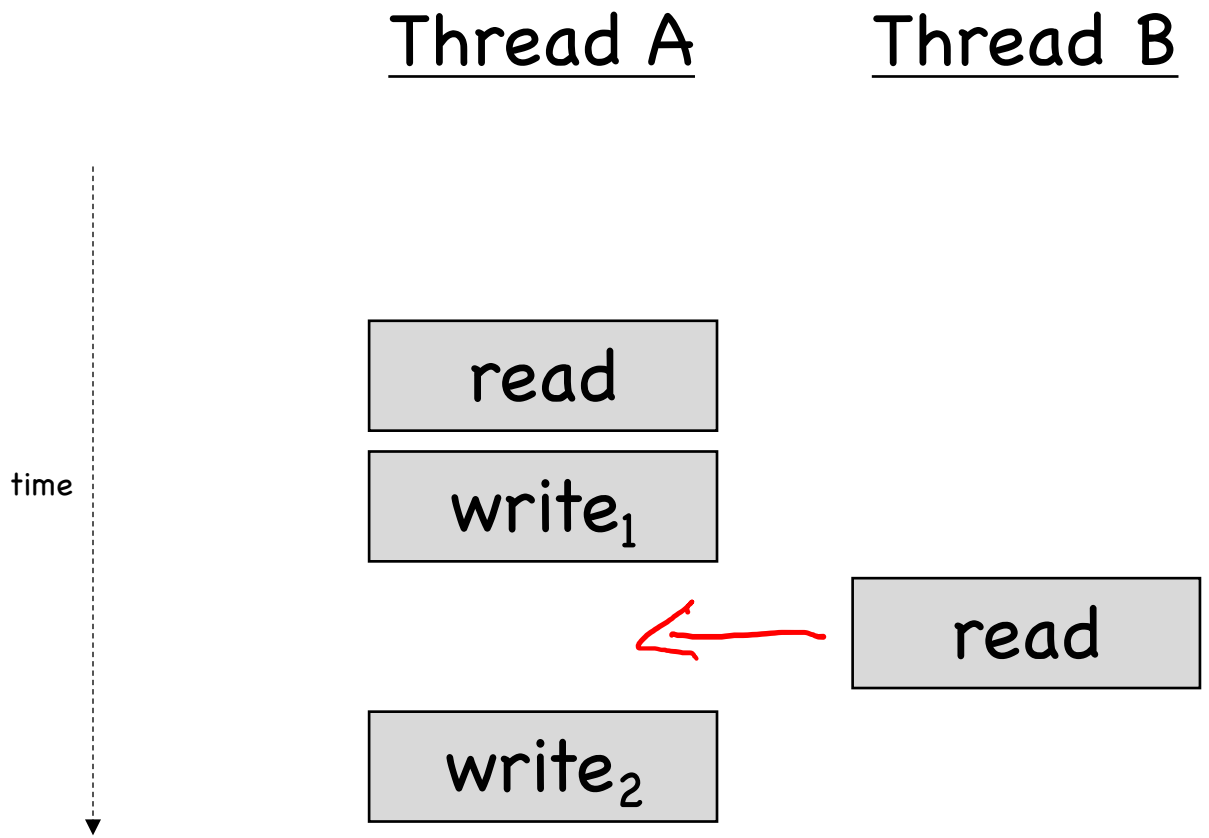


```
while(test-and-set(flag)) {}

Critical Section

flag = FALSE;
```

Thread Synchronization

Another Example:
Suppose 2 threads want to access the same variable...

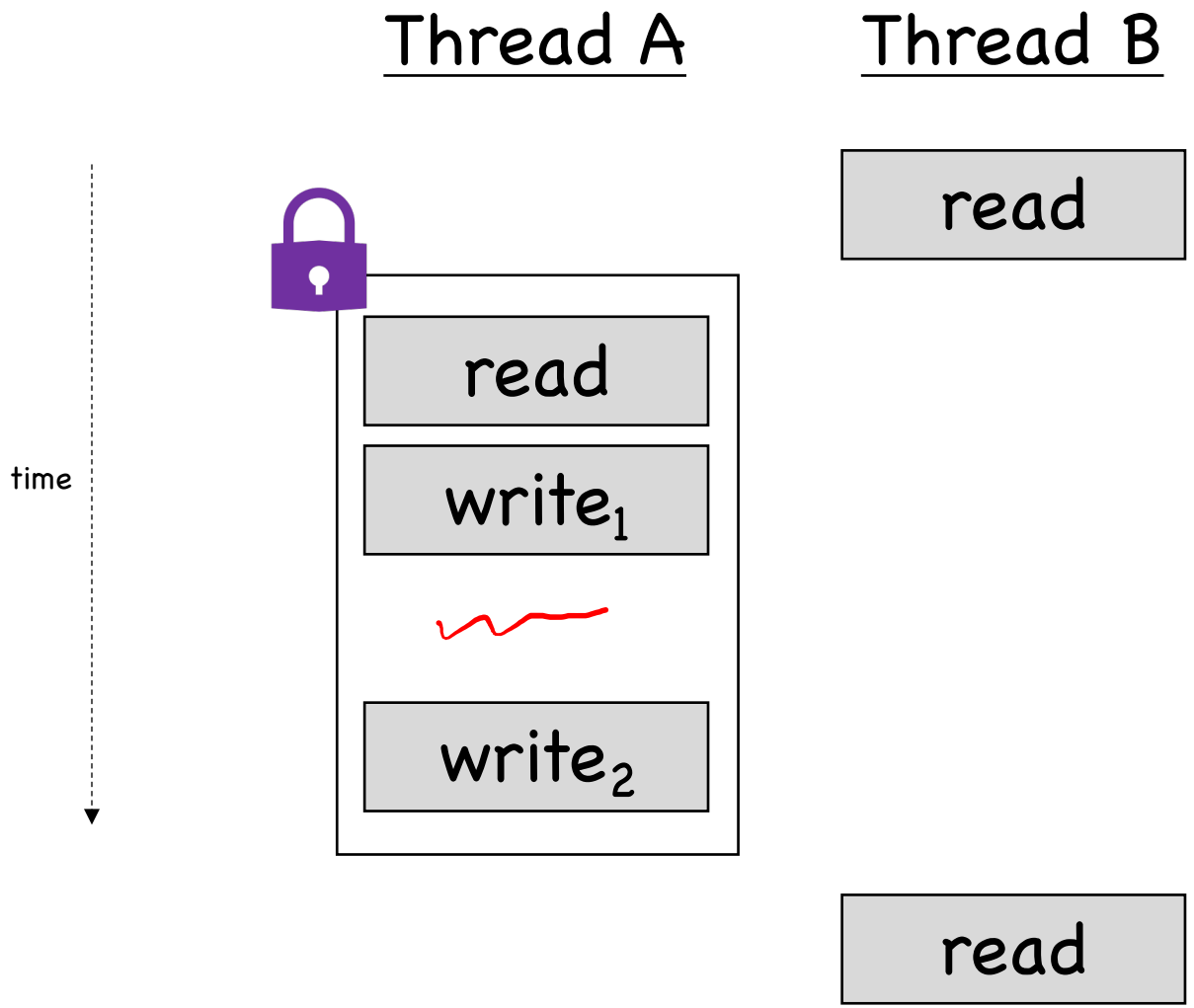
Thread Synchronization



Another Example:
Suppose 2 threads want to access the same variable...

Don't want Thread B to read *while* Thread A is writing...

Thread Synchronization



Another Example:
Suppose 2 threads want to access the same variable...

"Protect" Thread A's mult-step write operation.

(prevents simultaneous read/write of the shared variable)