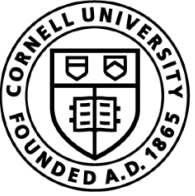


Cornell Bowers CIS
Computer Science

CS3410: Computer Systems and Organization

LEC23: Threads

Dr. Kevin Laeufer
Monday, November 17, 2025

Credits: Alvisi, Bala, Bracy, Garcia, Guidi, Kao, Laeufer, Martin, McKee, Sampson, Sirer, Van Renesse, Weatherspoon

Kevin Laeufer (he/him)
Visiting Lecturer

Hometown: Ithaca, NY

Ask me about: research, bike commuting, drywall mudding

Open OH: Monday 3pm - 4pm, Gates 425 (11/17, 11/24, 12/1, 12/8)

1:1 OH: Book here

- Back for 5 more lectures.
- New **open** office hours.
- Prof. Guidi is at Supercomputing 2025.

CS3410: Look how far we have come!

- Numbers: base conversion, binary addition, floating point ✅
- C Programming ✅
- Logic Gates, State (Registers and Latches), FemtoProc ✅
- RISC-V Assembly ✅
- Caches ✅
- Operating System: Processes, System Calls, Virtual Memory ✅
- Parallel Programming: Threads, Synchronization: **Next 3 Lectures**
- Special Topics: **12/1 and 12/3**
- Review: **12/8**

Plan for the next 3 lectures

- Threads:
  - Applications
  - Implementation
- Race Conditions
- Synchronization:
  - LR/SC
  - Atomic Increment,
  - Critical Section
  - Mutex

Threads

Introducing: Thread-Level parallelism

Threads are **separate tasks** within the same process. They:

- Share: code, data, heap, files
- Do not share: registers or stack

Prior Knowledge from CS2110

What is true about threads?

PollEv.com/cs3410

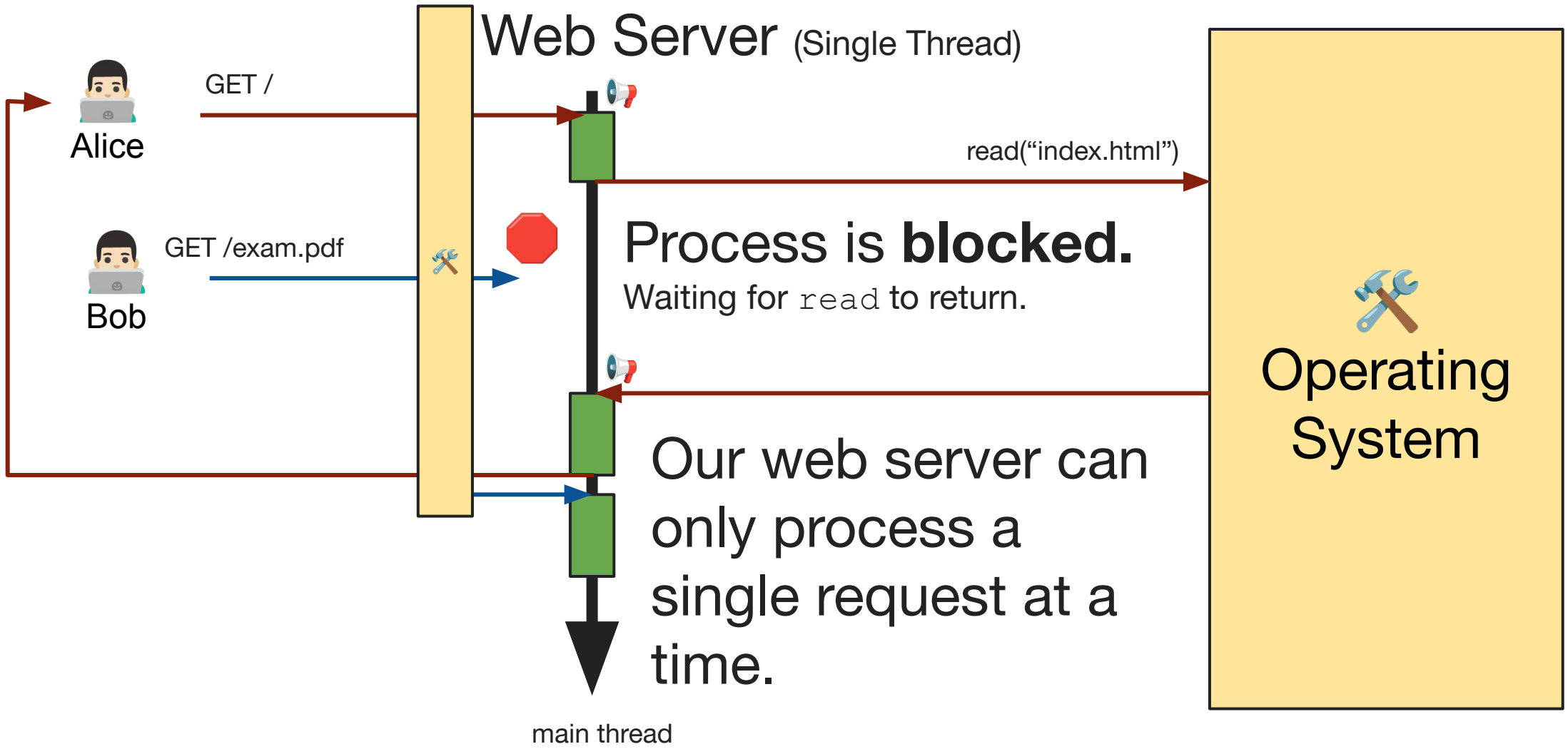
Process Question

How does control pass from a process to the operating system?

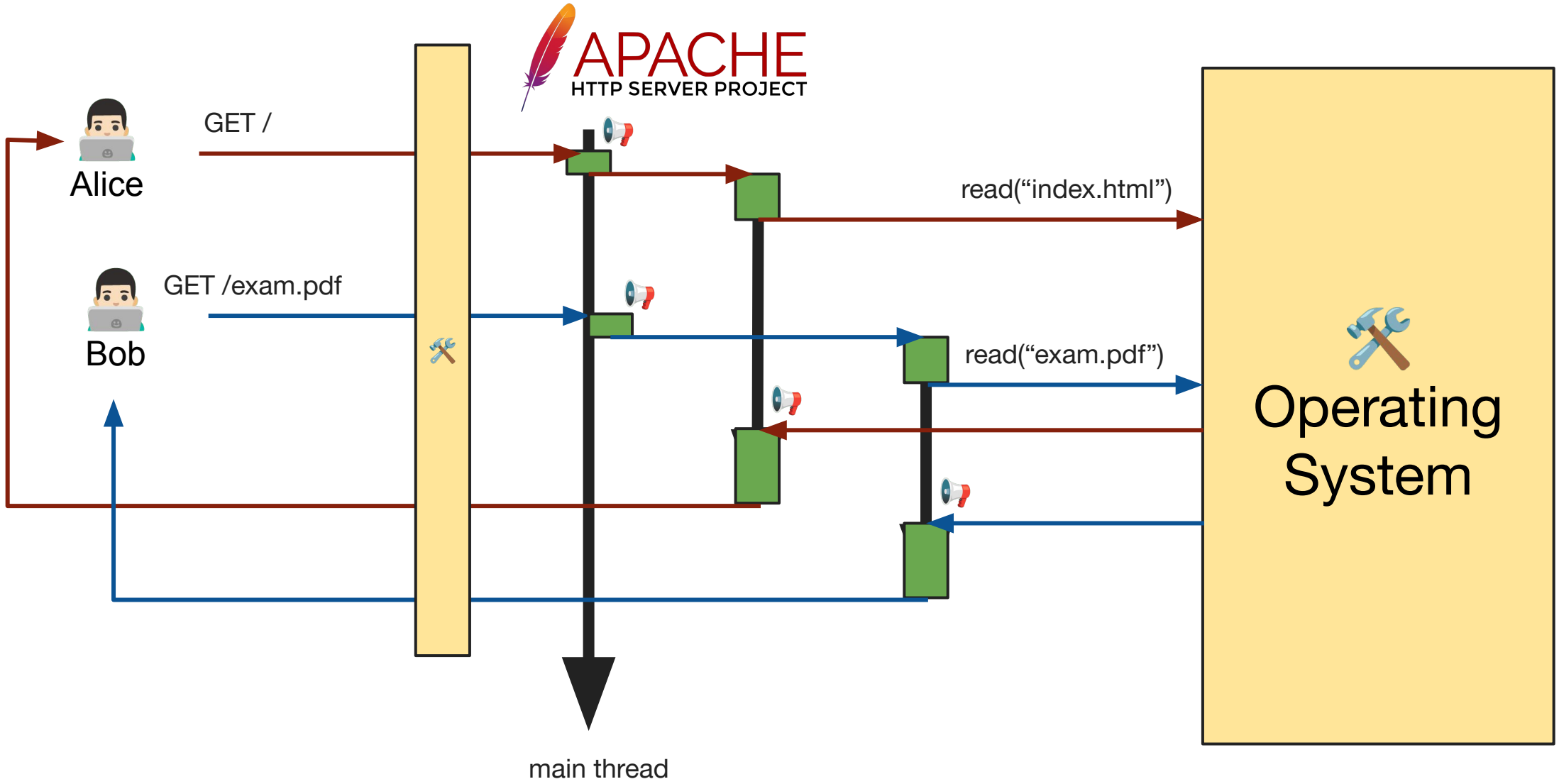
- You can only respond once.
- Try keeping your answer short. Ideally, a single **keyword**.

PollEv.com/cs3410

Case Study: Web Server



Case Study: Web Server



Threads for Web Servers

- Web servers generally do **little computation** but **many I/O operations** (file system and network interactions).
- System calls **block** the calling process.
- Each thread is like a *mini process* that can wait for a system call.
- **Shared data** in a web server:
  logs, authentication, HTTPS certificate

Case Study: Sieve of Eratosthenes

```
for (mp = start; mp < 10000; ++mp) {
 if (!get(mp)) { // mp is prime
   for (multiple = mp;
        multiple < 100000000; multiple += mp) {
     // multiple is not prime
     if (!get(multiple)) { set(multiple); }
   }
 }
}
```

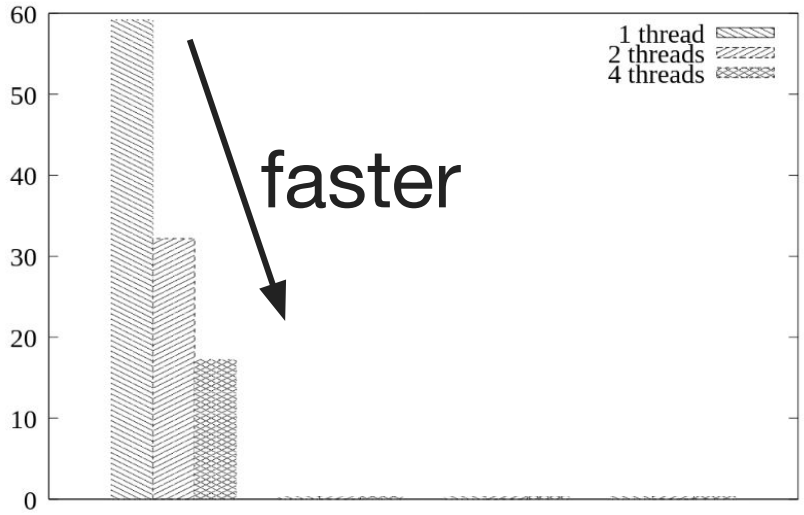
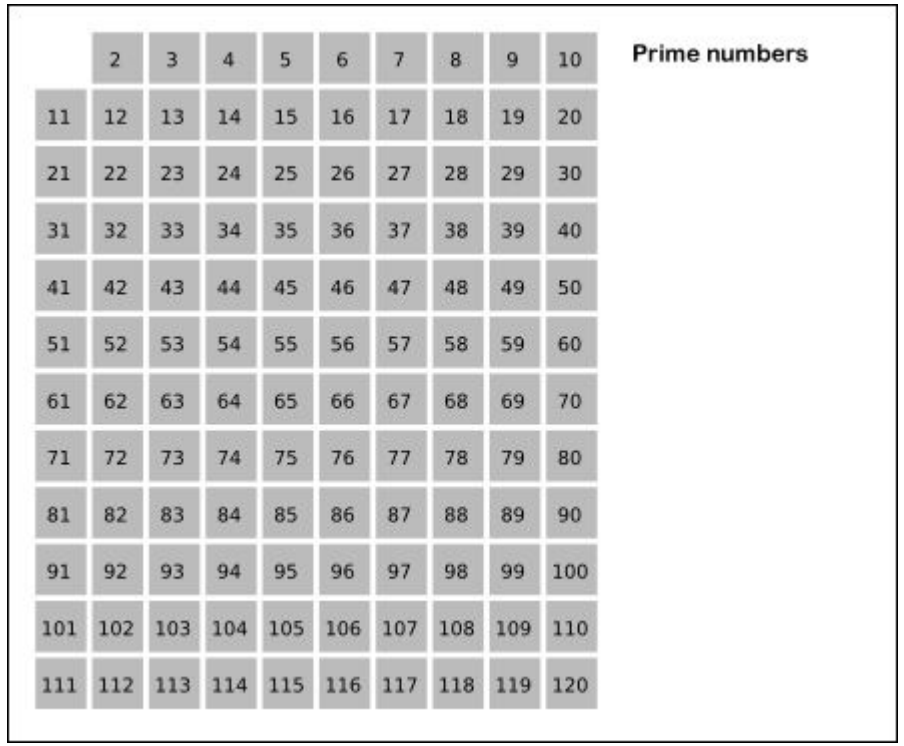


Figure 1. Sieve execution time for byte array (secs)



Execution on multiple cores leads to faster computation.

source: https://dl.acm.org/doi/pdf/10.1145/1065010.1065042

Threads for Compute Performance

- Without threads, a single process can only run on a single CPU core.
- Each thread can run on a different CPU core.
- If compute and not I/O is the main concern, this is the rule of thumb: Use **as many threads as CPU cores** to optimally balance thread overhead and compute throughput.

Threads vs Processes

Threads

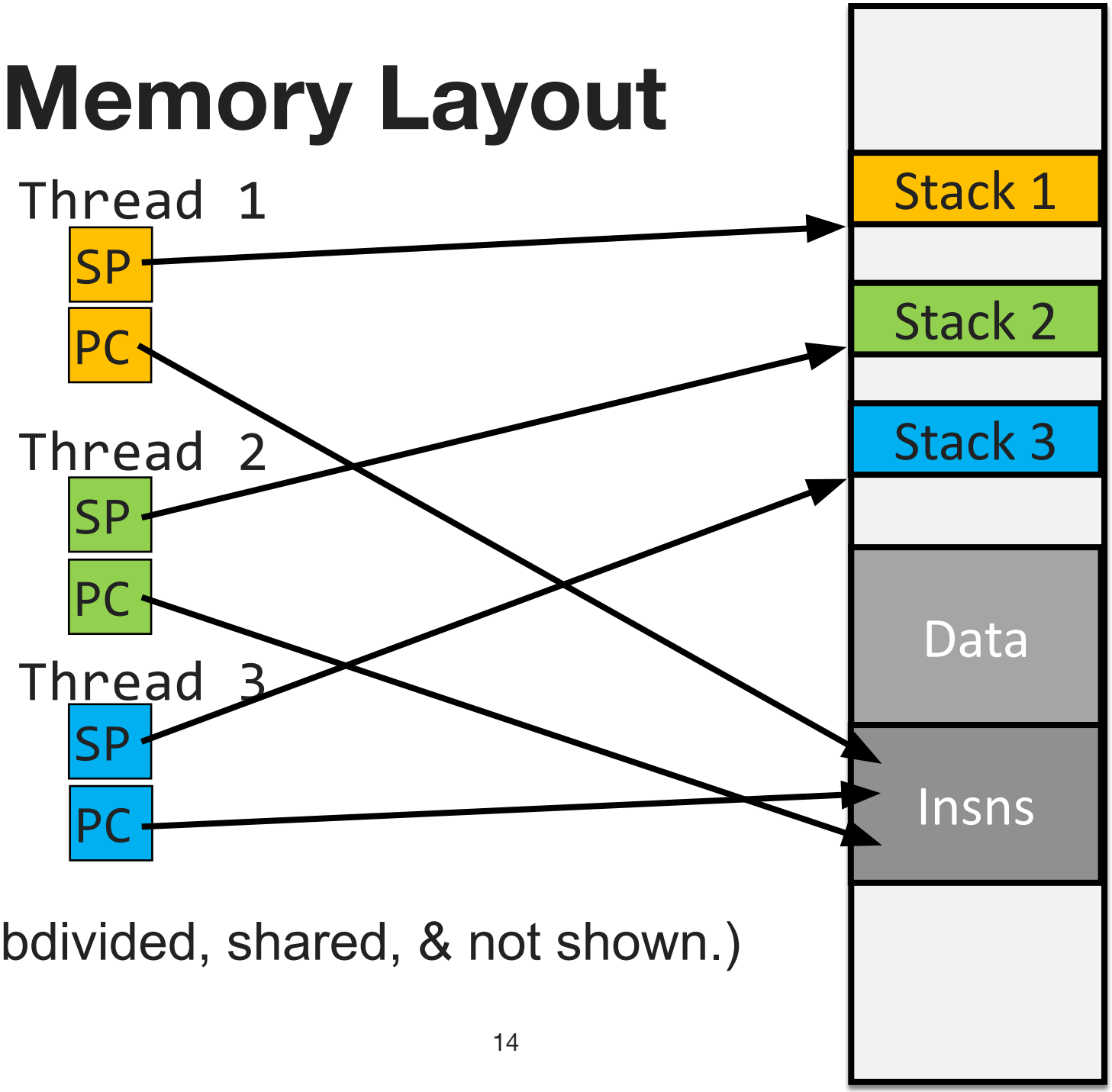
- API: create + join
- private:
  - Stack, SP, PC, registers
- shared
  - address space, files

Processes

- API: fork (and exec) + wait
- private
  - SP, PC, registers, address space

Shared address space means that we can use the same page table. Thus, **context switching between threads is faster** than between processes.

Thread Memory Layout



(Heap subdivided, shared, & not shown.)

Aside: OS- vs. User-level Threads

Can we implement threads as a library *without* operating system support?

Provided by the OS:

- Context switch on system call or preemption

User-level thread challenges:

- Allocate memory for stack? → malloc
- Context switch?:
  - Cannot preempt
  - Thread must yield
- A single system call blocks all threads → non-blocking system calls
- A single process can only use a single CPU core →
  focus on I/O heavy workloads or map user level threads to a limited number of OS threads

Why (OS-level) Threads?

Performance: exploiting multiple processors

Do threads make sense on a single core?

Responsiveness

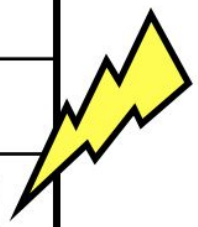
- threads can do work in the background

Mask long latency of I/O devices

- do useful work while waiting

Non-Atomicity: Case Study #1

Time	You	Your roomate
3:00	Arrive home	
3:05	Check fridge → no milk	
3:10	Leave for grocery	
3:15		Arrive home
3:20	Buy milk	Check fridge → no milk
3:25	Arrive home, milk in fridge	Leave for grocery
3:30		
3:35		Buy milk
3:40		Arrive home, milk in fridge!



Non-Atomicity: Case Study #2

```
Thread 1                    Thread 2
for(i=0; i<5;i++){          for(i=0; i<5;i++) {
   x++                         x++
}                           }
```

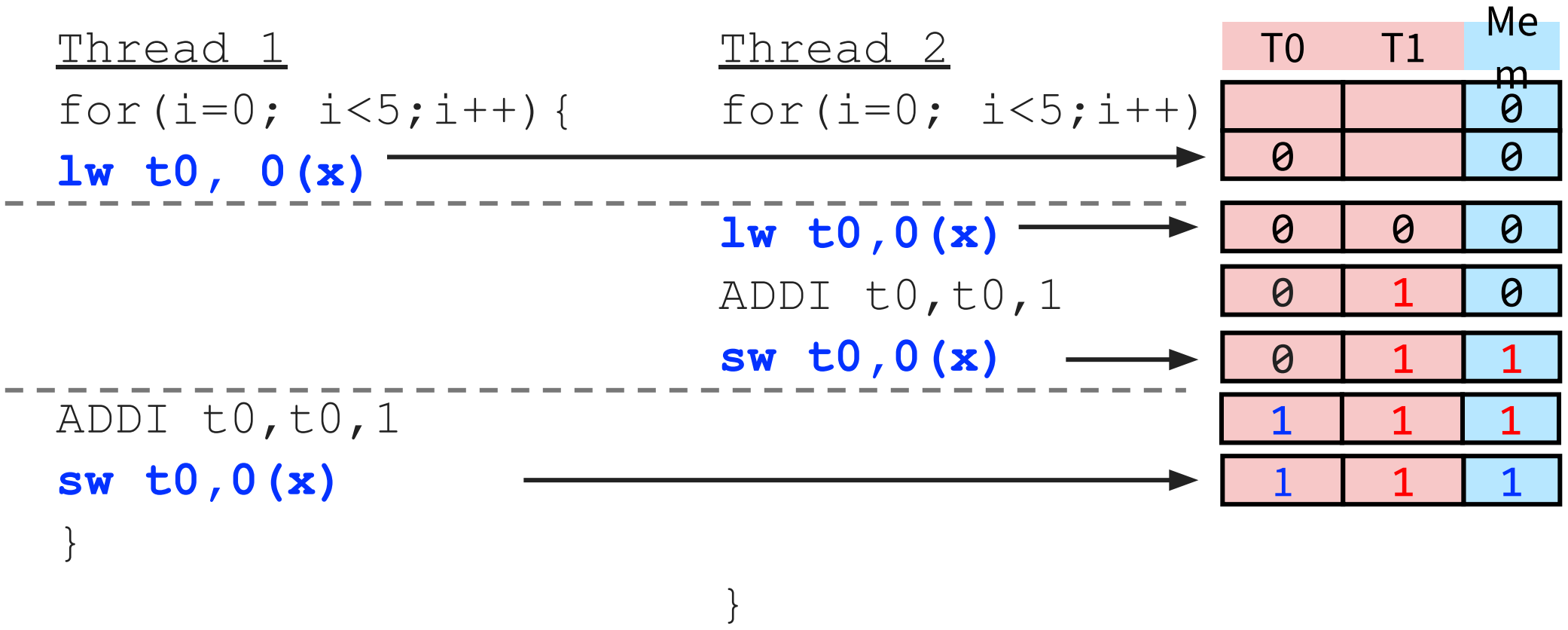
Assume `x` is a global variable in the Data Segment, initialized to 0.

What will the value of `x` be after both loops finish?

a) 5
b) 8
c) 10
d) Could be any of the above
e) Could be any value

PollEv.com/cs3410

Non-Atomicity: Case Study #2

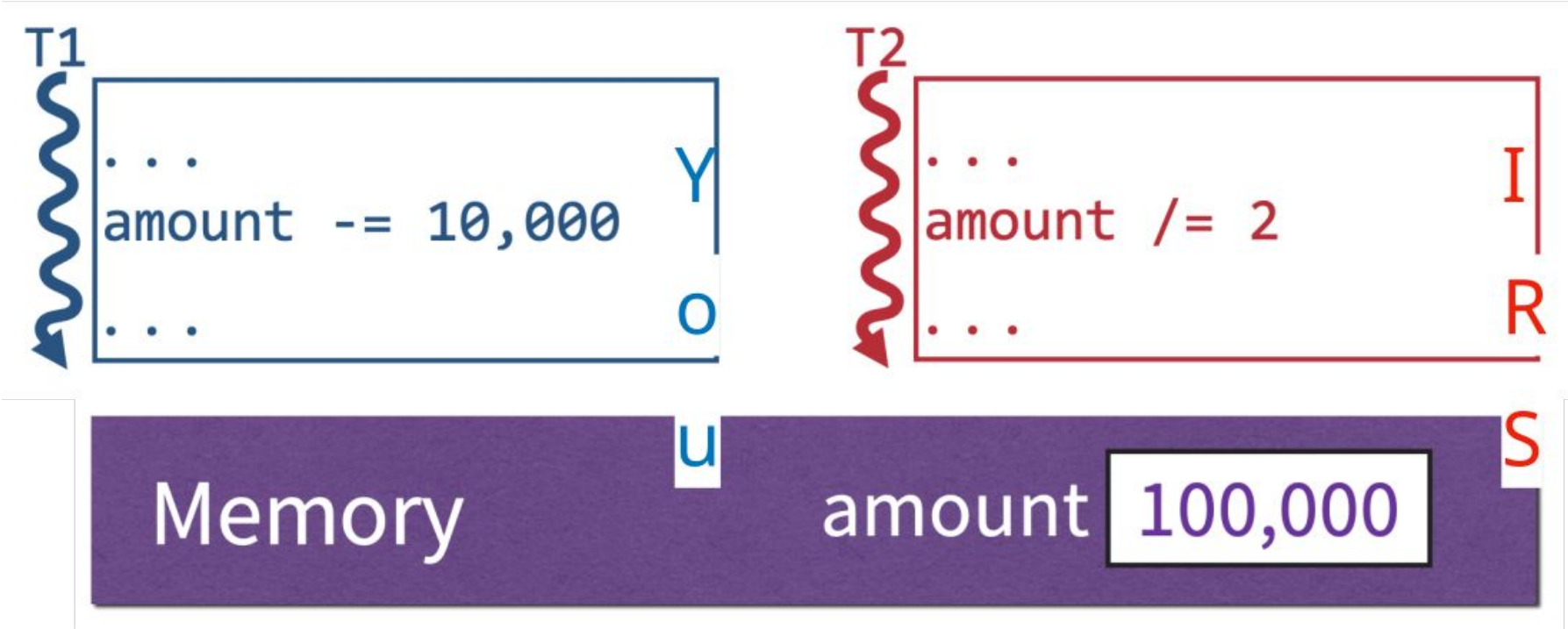


Advanced Note: on multiple CPU cores, without *Sequential Consistency* guarantees effects are even harder to explain.

Non-Atomicity: Case Study #3

Consider two threads updating a shared variable **amount**:

- One thread (you) wants to decrement amount by $10K
- Other thread (IRS) wants to decrement amount by 50%

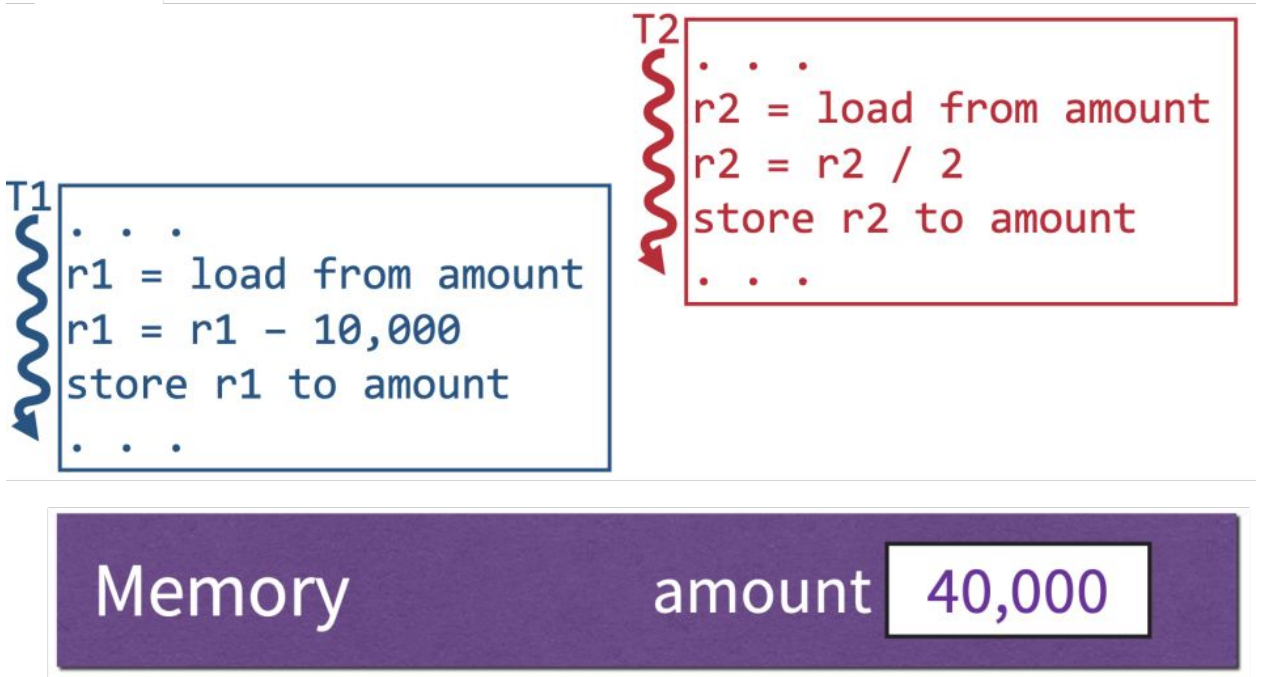


Non-Atomicity: Case Study #3

We decompose the high-level C math into pseudo assembly.

This ordering works:

- $100,000
- $50,000
- $40,000

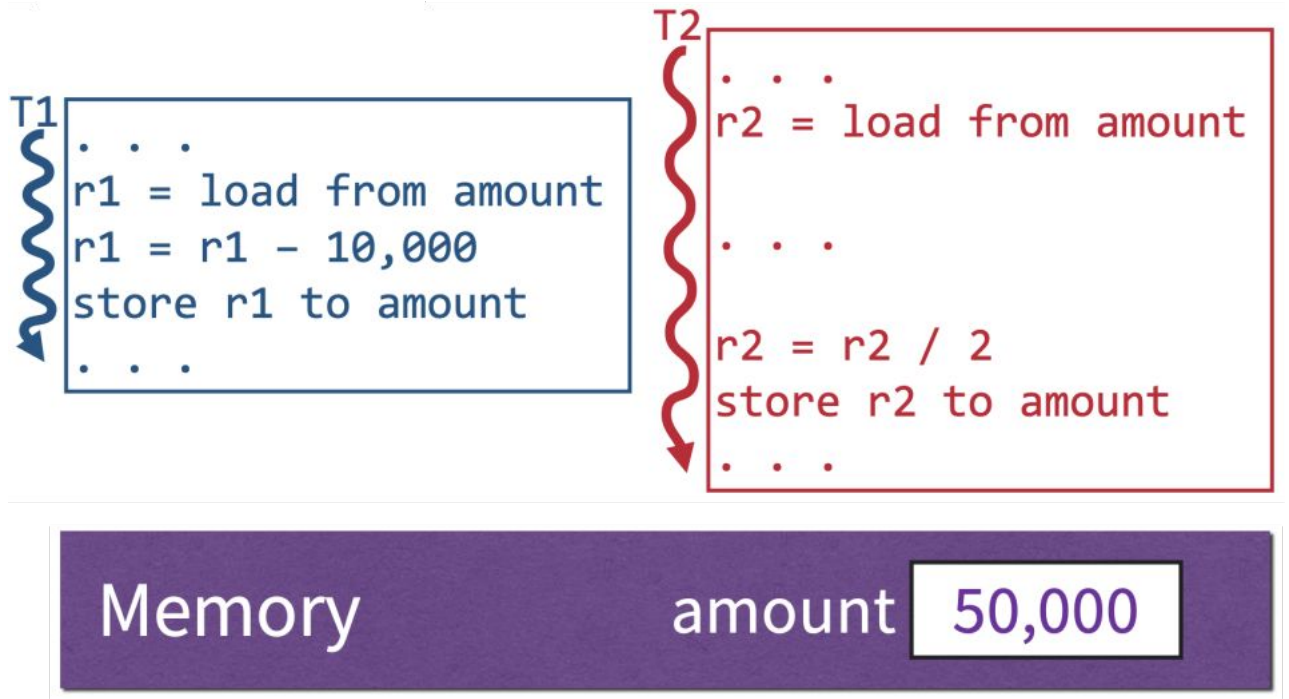


Non-Atomicity: Case Study #3

We decompose the high-level C math into pseudo assembly.

Now we **lose an update:**

- $100,000
- $50,000

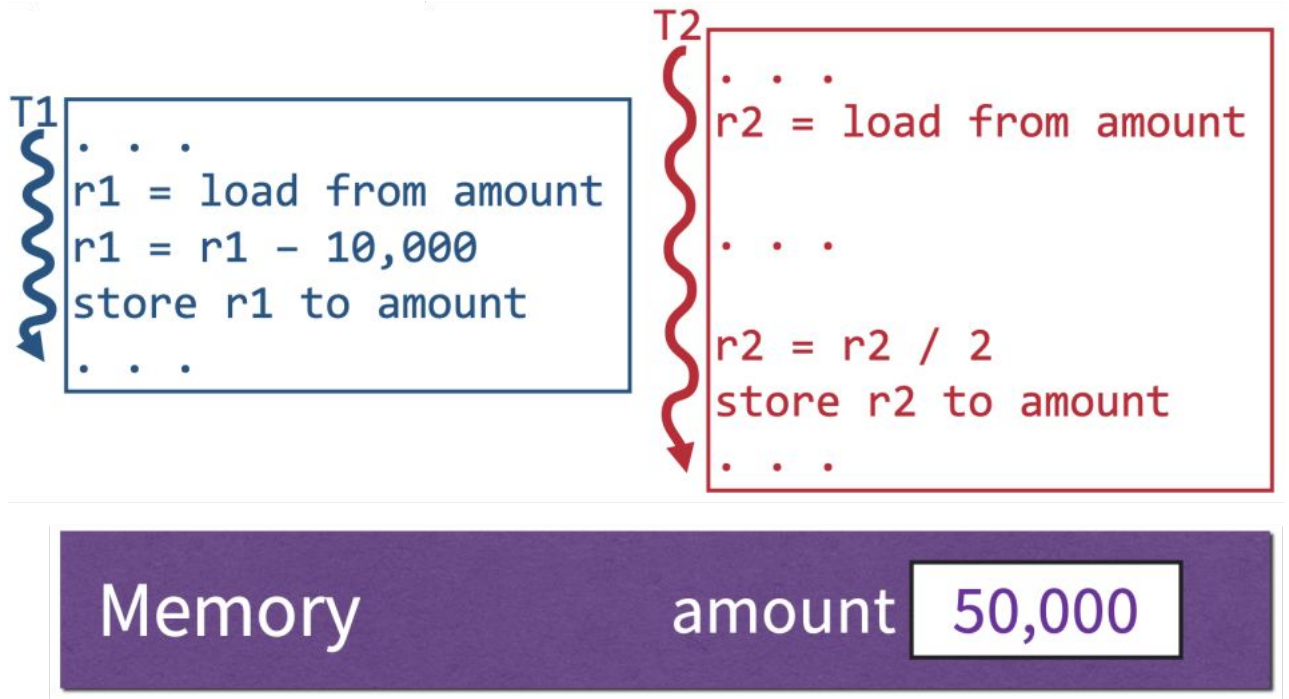


Non-Atomicity: Case Study #3

We decompose the high-level C math into pseudo assembly.

Now we **lose an update:**

- $100,000
- $50,000



Big Picture: Programming with threads

Within a thread: execution is sequential

Between threads?

- (Almost) no ordering or timing guarantees
- Might all execute on same core, or not!

Problem: difficult to program, difficult to reason about

- Behavior can depend on subtle timing differences
- Bugs may be impossible to reproduce

Race conditions

Occur when two threads are accessing the same memory location and at least one access is a write.
(two concurrent reads are fine!)

Challenges of Race Conditions

- Races are intermittent, may occur rarely or in certain scenarios
- They often appear to happen ***randomly***

Program is correct *only* if *all possible* schedules are safe

Example Race Condition: 2 threads trying to increment `x`

Critical Sections

- To eliminate races: identify *Critical Sections*
  - Places in code where shared state is read and written
  - ***Only 1 thread gets to execute at a time***
  - Others *wait their turn*

```
...
tmp = *cur_score;                                  //read shared var
*cur_score = update_score(tmp);                    //write shared var
if (tmp > *high_score) { *high_score = tmp; } //r/w shared var
...
```

Critical Section!

Critical Sections

- To eliminate races: identify *Critical Sections*
  - Places in code where shared state is read and written
  - ***Only 1 thread gets to execute at a time***
  - Others *wait their turn*

```
cs_enter();
tmp = *cur_score;                                  //read shared var
*cur_score = update_score(tmp);                    //write shared var
if (tmp > *high_score) { *high_score = tmp; } //r/w shared var
cs_exit();
```

Today

- Threads represent concurrent tasks within a process:
  - **shared:** memory + address space, file descriptors
  - **separate:** PC, stack, register values
- Advantages:
  - Perform work during **blocking I/O**
  - Utilize more than one CPU core from a single process.
- Problem:
  - Programmer must synchronize explicitly
  - Race conditions introduce non-determinism → hard to catch bugs!