

Concurrency Overview

CS 475: Concurrent & Distributed Systems (Fall 2021)

Lecture 2

Yue Cheng

Concurrency

- Process vs. thread
- Race conditions
- Locks
- Concurrency in Go

What is a process?

What is a process?

- **Programs** are code (static entity)
- **Processes** are running programs

- Java analogy
 - class -> “program”
 - object -> “process”

What is in a process?

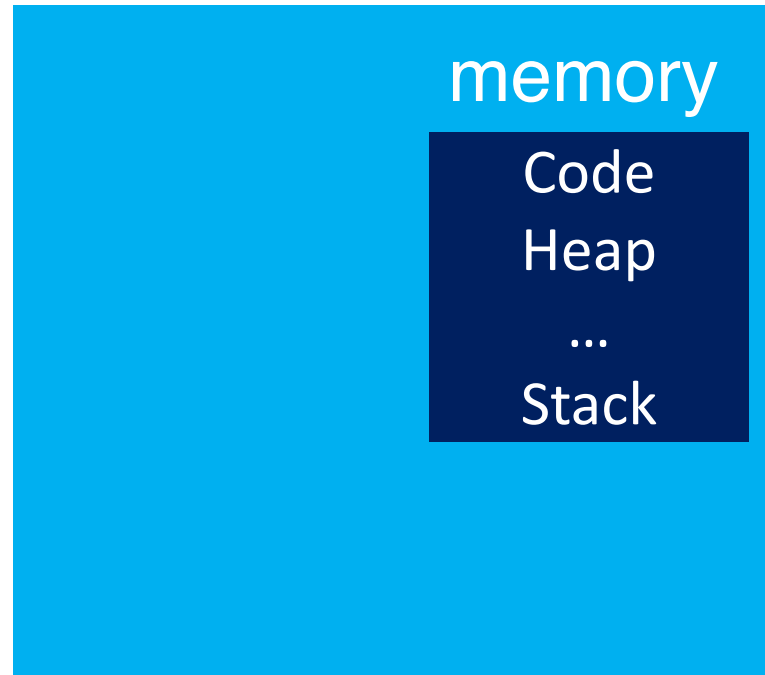
Process



What things change as a program runs?

What is in a process?

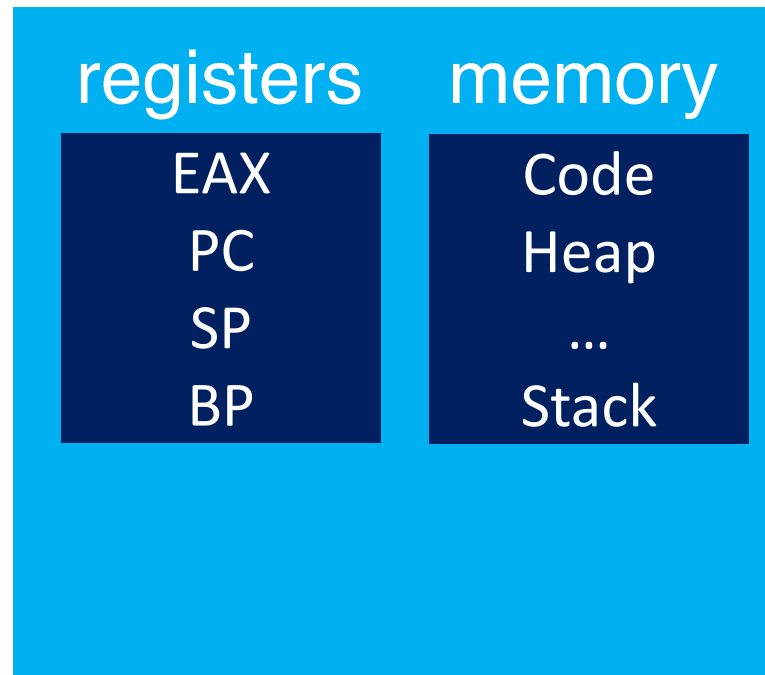
Process



What things change as a program runs?

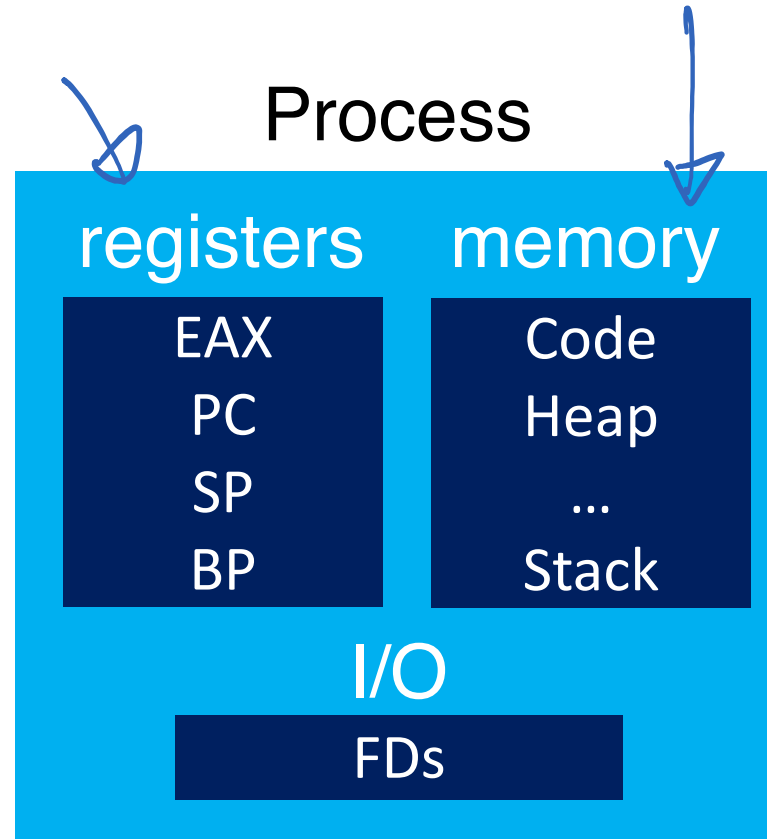
What is in a process?

Process



What things change as a program runs?

What is in a process?



What things change as a program runs?

Peeking inside

- Processes share code, but each has its own “context”
- CPU
 - Instruction pointer (Program Counter)
 - Stack pointer
- Memory
 - Set of memory addresses (“address space”)
 - `cat /proc/<PID>/maps`
- Disk
 - Set of file descriptors
 - `cat /proc/<PID>/fdinfo/*`

Threads

Why thread abstraction?

Process abstraction: Challenge 1

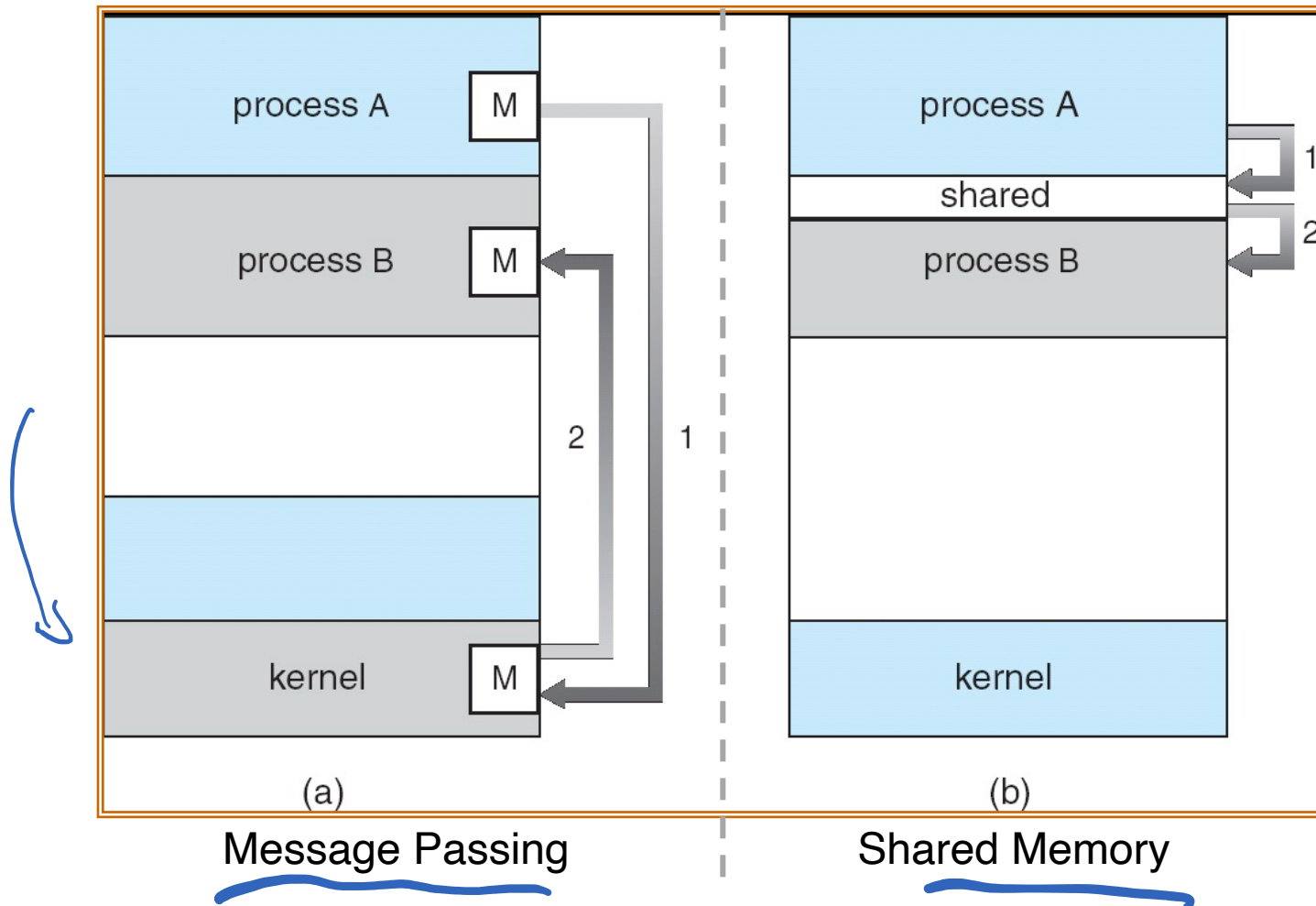
- Inter-process communication (IPC)

Inter-process communication

- Mechanism for processes to communicate and to synchronize their actions
- Two models
 - Communication through a shared memory region
 - Communication through message passing



Communication models



Communication through message passing

- Message system – processes communicate with each other **without** resorting to shared variables
- A message-passing facility must provide at least two operations:
 - `send(message, recipient)`
 - `receive(message, sender)`
- With **indirect** communication, the messages are sent to and received from **mailboxes** (or, **ports**)
 - `send(A, message) /* A is a mailbox */`
 - `receive(A, message)`



Communication through message passing

- Message passing can be either **blocking** (**synchronous**) or **non-blocking** (**asynchronous**)
 - **Blocking Send:** The sending process is blocked until the message is received by the receiving process or by the mailbox
 - **Non-blocking Send:** The sending process resumes the operation as soon as the message is received by the kernel
 - **Blocking Receive:** The receiver blocks until the message is available
 - **Non-blocking Receive:** “Receive” operation does not block; it either returns a valid message or a default value (null) to indicate a non-existing message

Communication through shared memory

- The memory region to be shared must be explicitly defined
- System calls (Linux):
 - `shmget` creates a shared memory block
 - `shmat` maps/attaches an existing shared memory block into a process's address space
 - `shmdt` removes (“unmaps”) a shared memory block from the process's address space
 - `shmctl` is a general-purpose function allowing various operations on the shared block (receive information about the block, set the permissions, lock in memory, ...)
- Problems with `simultaneous access` to the shared variables

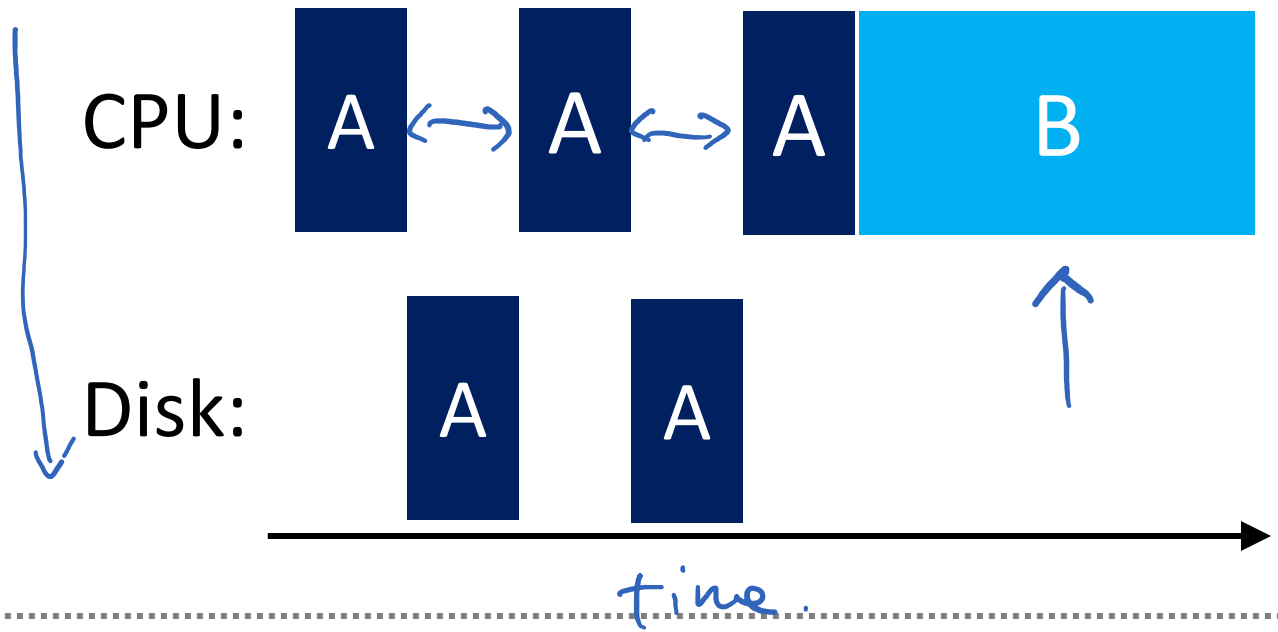
Process abstraction: Challenge 1

- Inter-process communication (IPC)
 - Cumbersome programming!
 - • Copying overheads (inefficient communication)
 - Expensive context switching (why expensive?)

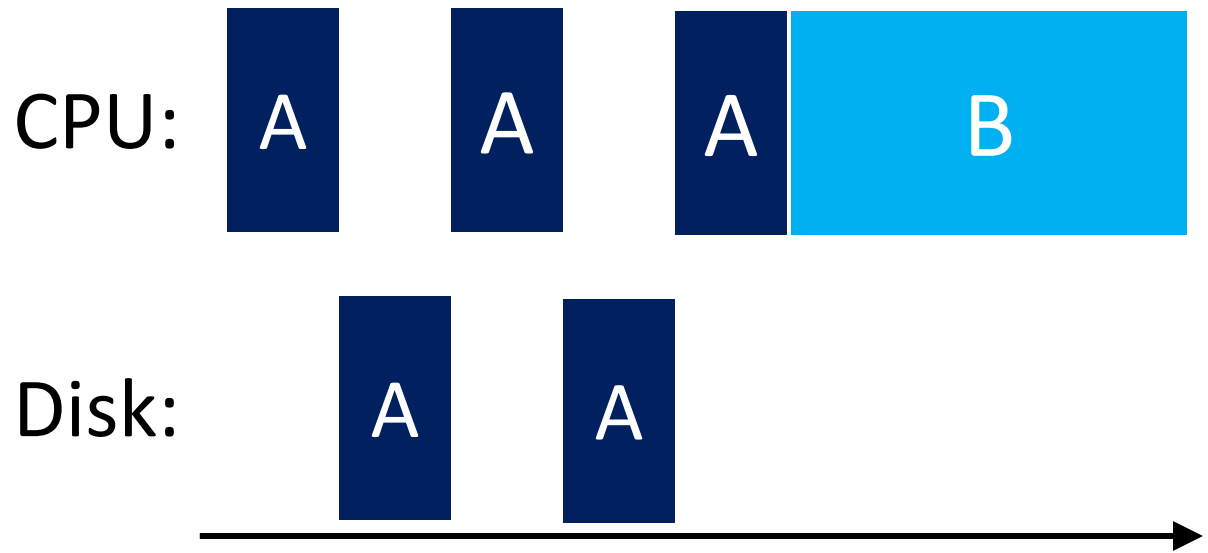
Process Abstraction: Challenge 2

- Inter-process communication (IPC)
 - Cumbersome programming!
 - Copying overheads (inefficient communication)
 - Expensive context switching (why expensive?)
- CPU utilization

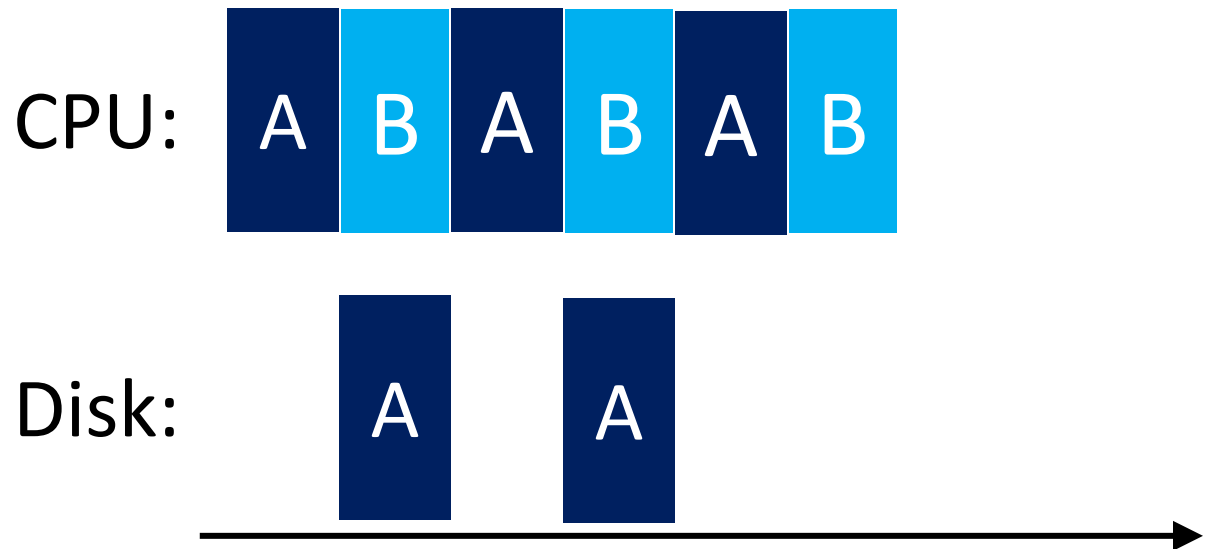
(a) Not interleaved



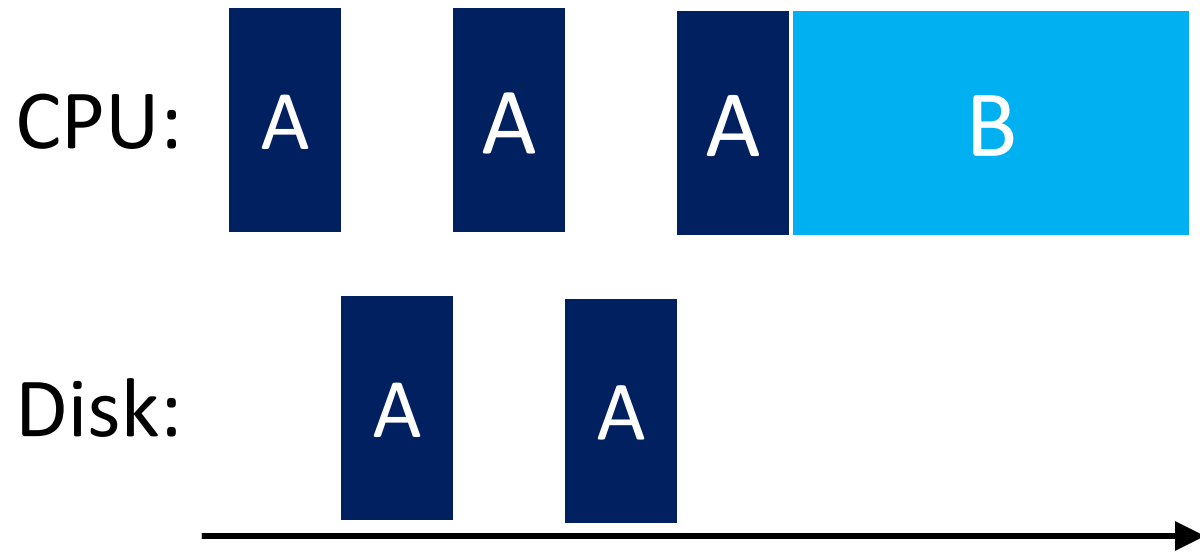
(a) Not interleaved



(b) Interleaved

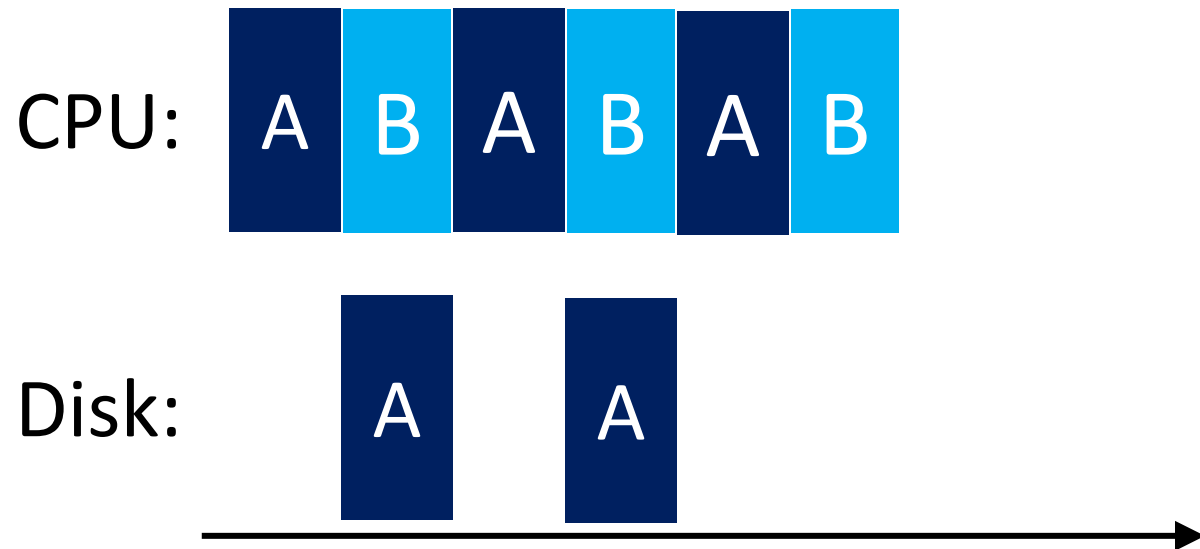


(a) Not interleaved



..... What if there is only one process?

(b) Interleaved



CPU trends – What Moore's Law implies...

- The future
 - Same CPU speed
 - More cores (to scale-up or scale-out)
- Faster programs => concurrent execution
- **Goal:** Write applications that fully utilize many CPU cores...

Goal

- Write applications that fully utilize many CPUs...

Strategy 1

- Build applications from many communication processes
 - Like Chrome (process per tab)
 - Communicate via `pipe()` or similar
- Pros/cons?

Strategy 1

- Build applications from many communication processes
 - Like Chrome (process per tab)
 - Communicate via `pipe()` or similar
- Pros/cons? – That we've talked about in previous slides
 - Pros:
 - Don't need new abstractions!
 - Better (fault) isolation?
 - Cons:
 - Cumbersome programming using IPC
 - Copying overheads
 - Expensive context switching

Strategy 2

- New abstraction: the **thread**

Introducing thread abstraction

- New abstraction: the **thread**
- Threads are just **like processes**, but threads **share the address space**

Thread

- A process, as defined so far, has only **one thread of execution**
- **Idea:** Allow multiple threads of **concurrently running** execution within the same process environment, **to a large degree independent** of each other
 - Each thread may be executing different code at the same time

Process vs. Thread

- Multiple threads within a process will share
 - The address space
 - Open files (file descriptors)
 - Other resources
- Thread
 - Efficient and fast resource sharing
 - Efficient utilization of many CPU cores with only one process
 - Less context switching overheads

CPU 1



CPU 2



CPU 1

Running
thread 1

PC

CPU 2

Running
thread 2

PC

CPU 1

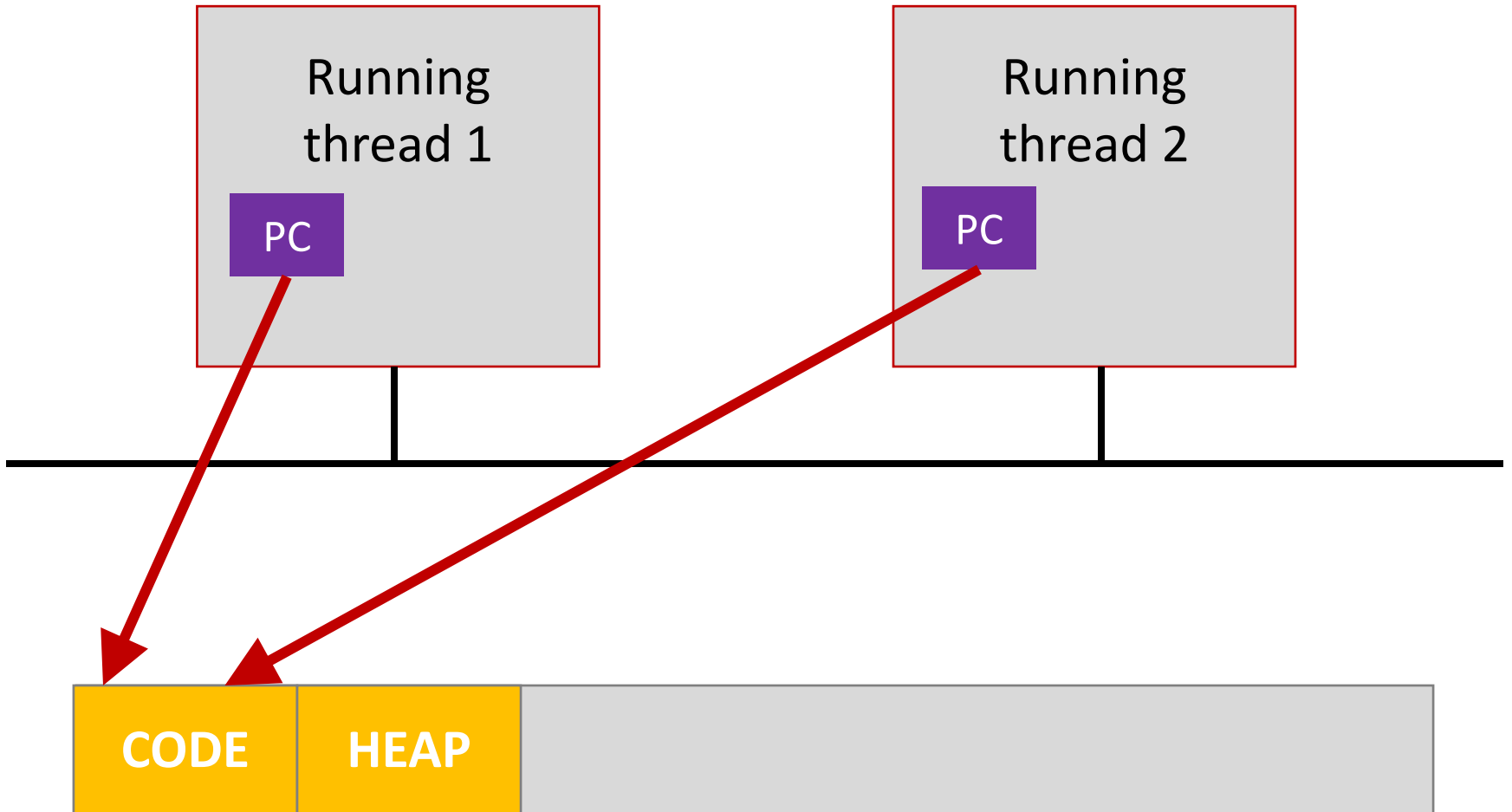
Running
thread 1

PC

CPU 2

Running
thread 2

PC



Virtual mem

CPU 1

Running
thread 1

PC

CPU 2

Running
thread 2

PC

Each thread may be executing
different code at the same time

CODE

HEAP

Virtual mem

CPU 1

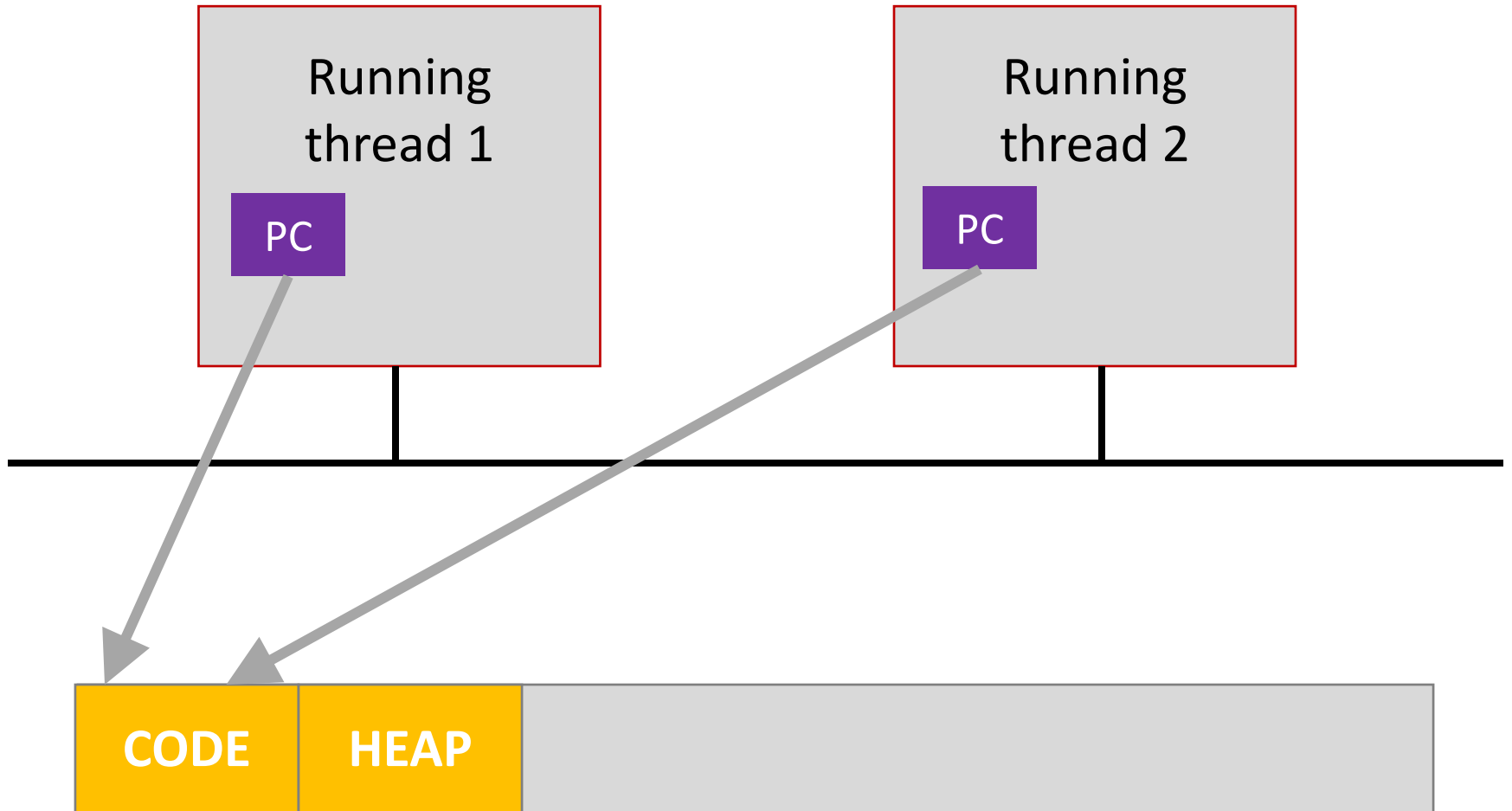
Running
thread 1

PC

CPU 2

Running
thread 2

PC



Virtual mem

CPU 1

Running
thread 1

PC

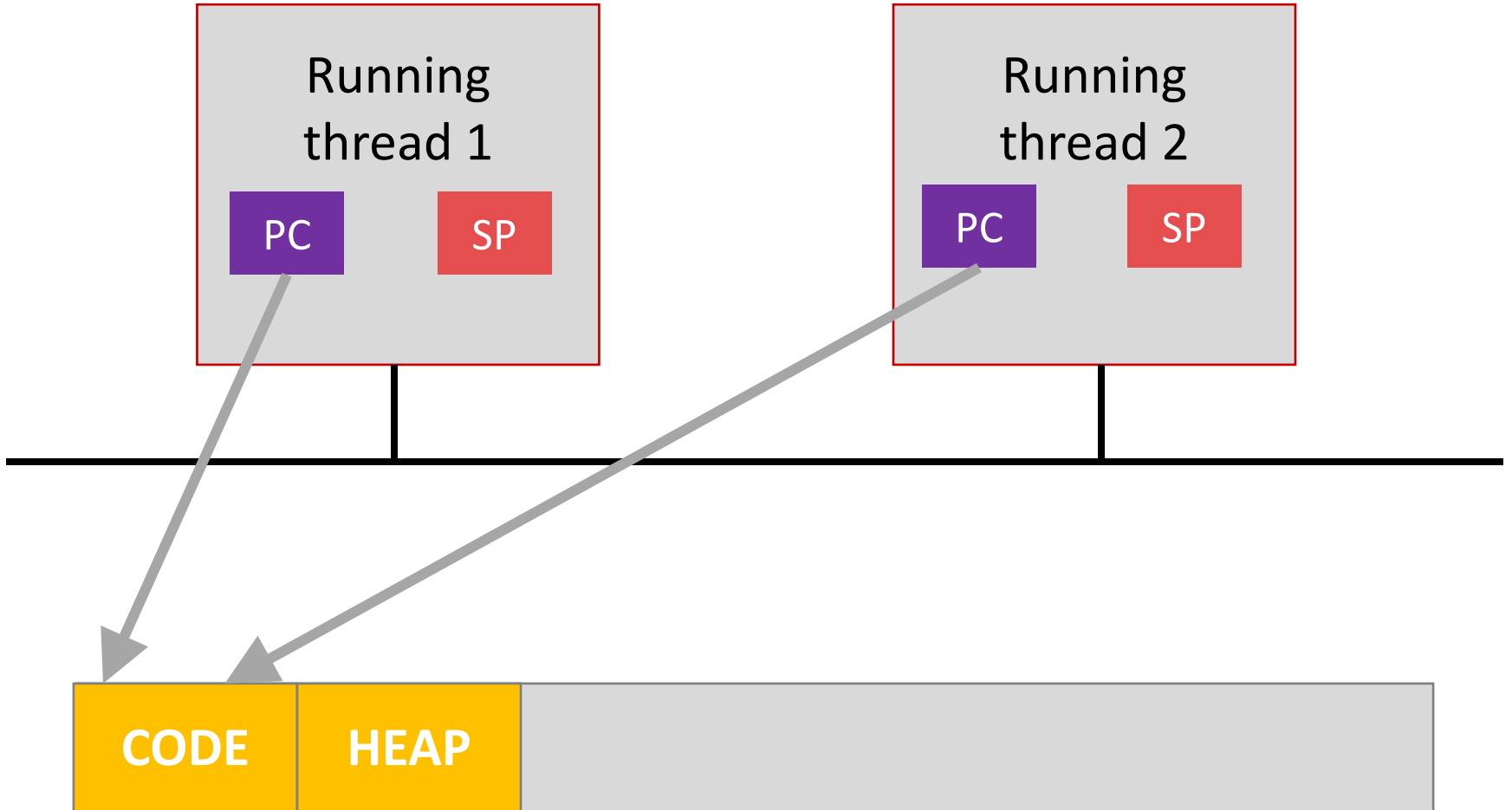
SP

CPU 2

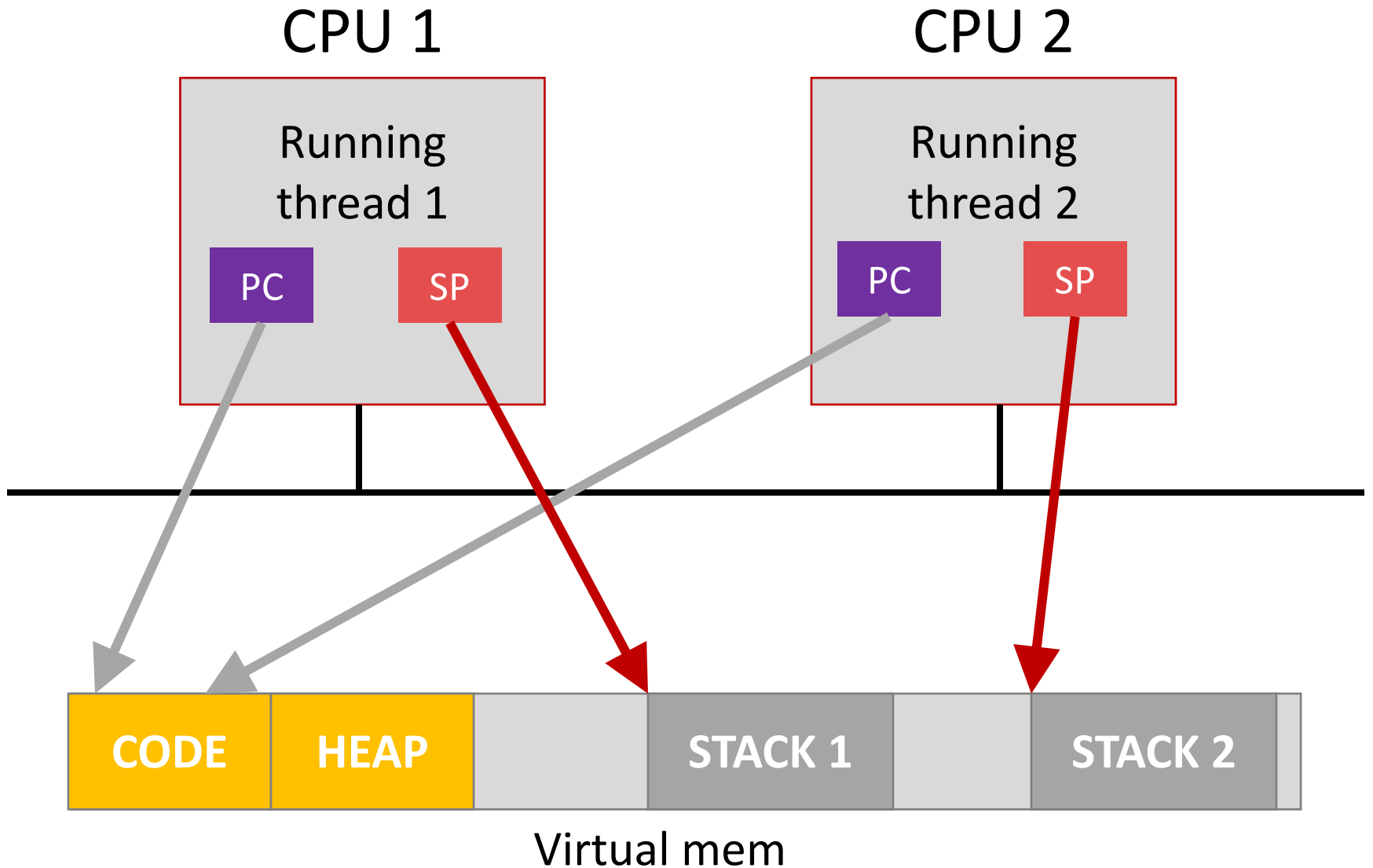
Running
thread 2

PC

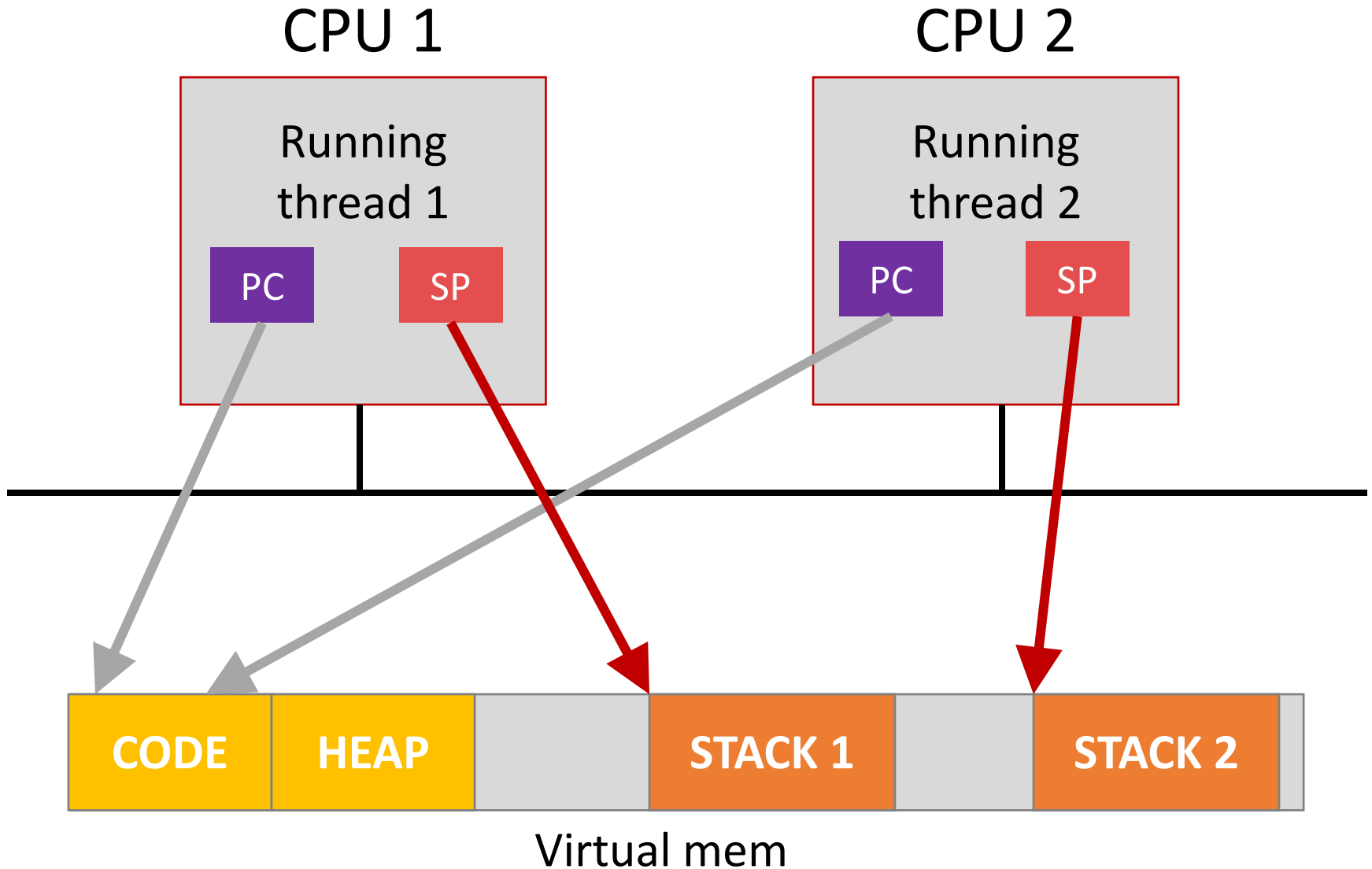
SP

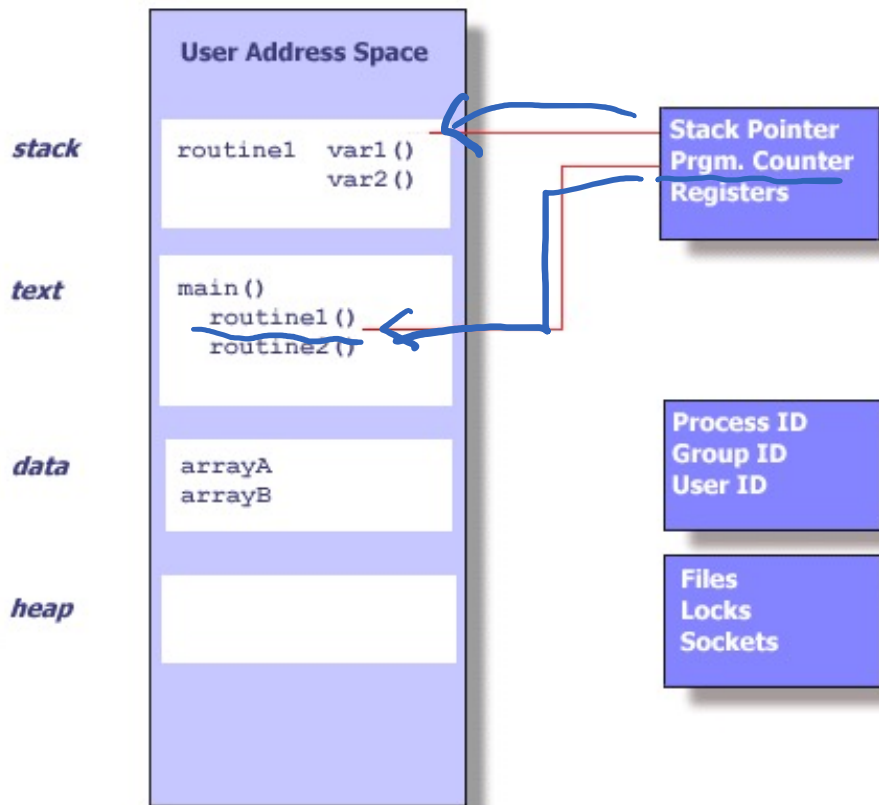


Virtual mem

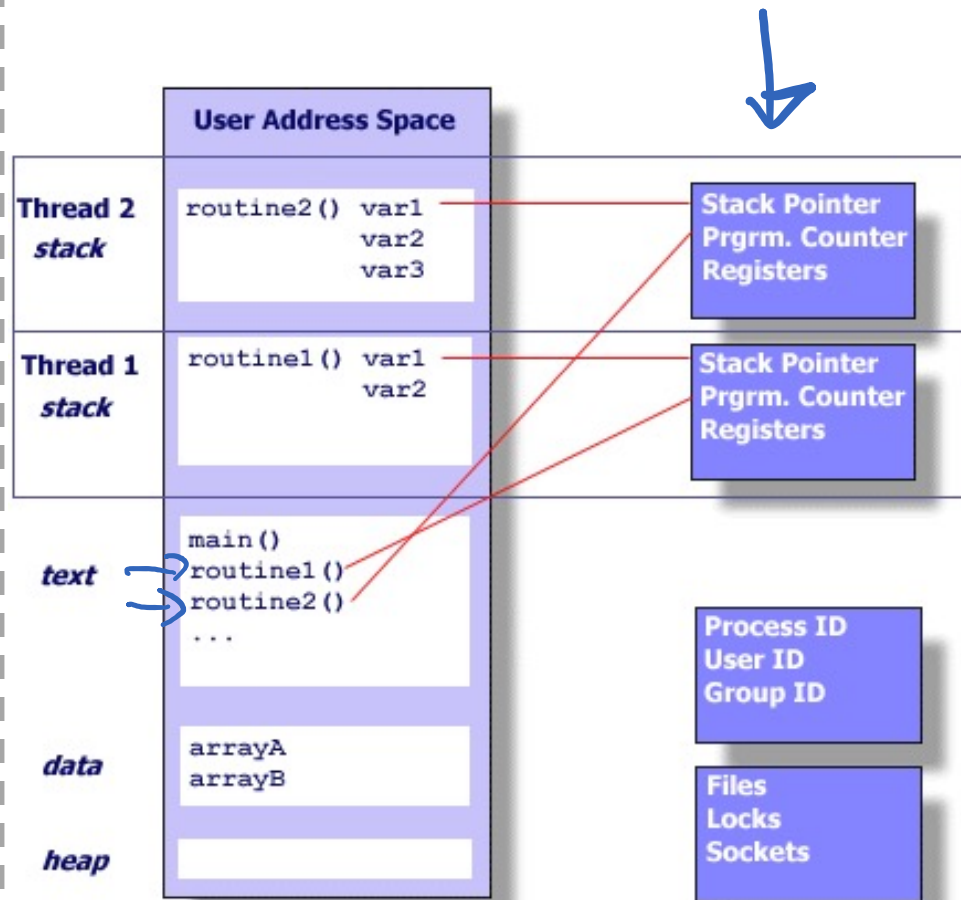


Thread executing **different functions** need **different stacks**





Linux process

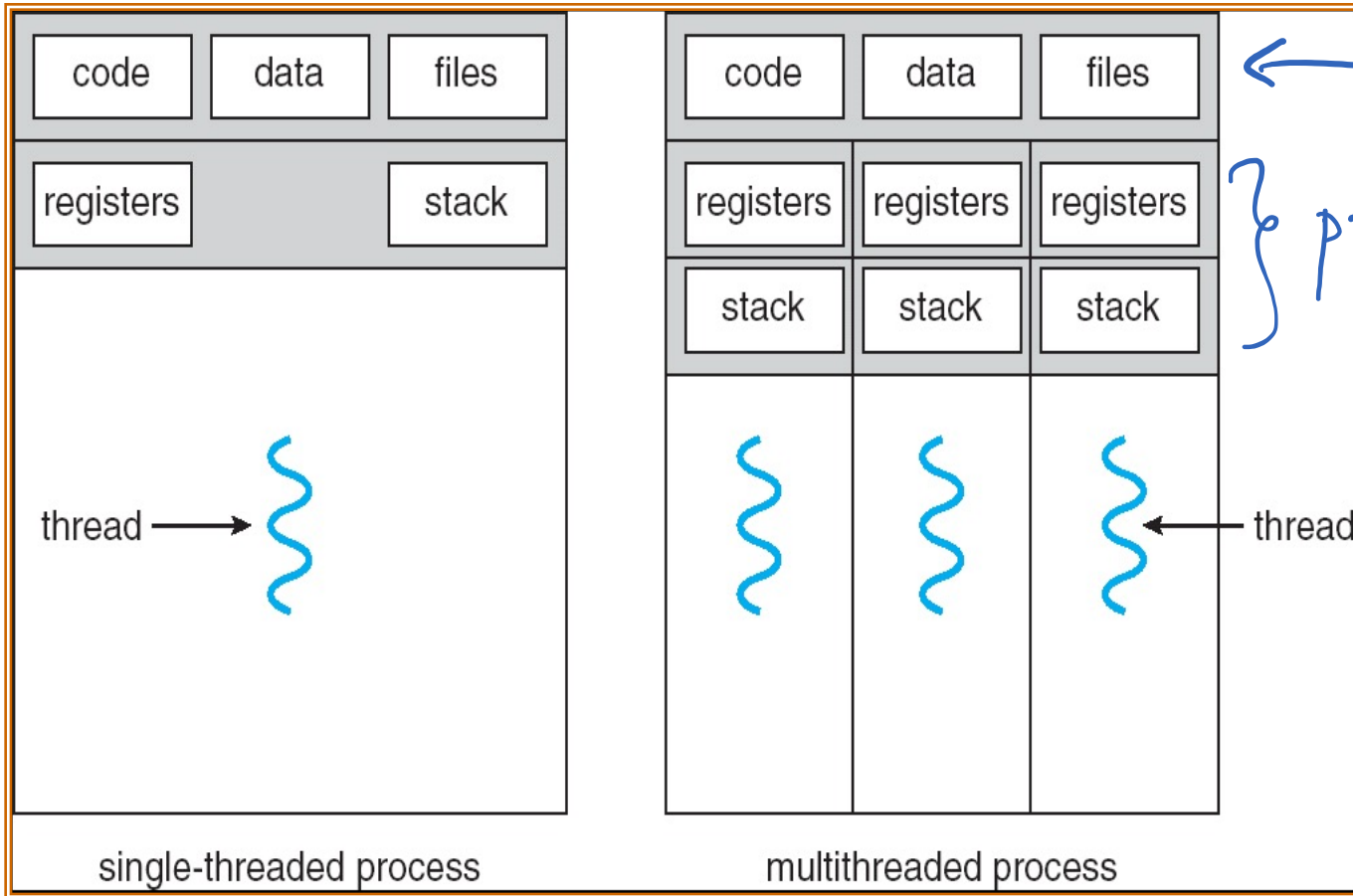


Threads within a Linux process

pthread

*: <https://computing.llnl.gov/tutorials/pthreads/>

Single- vs. Multi-threaded Process



Concurrency

- Process vs. thread
- Race conditions
- Locks
- Concurrency in Go

Example: Bank account

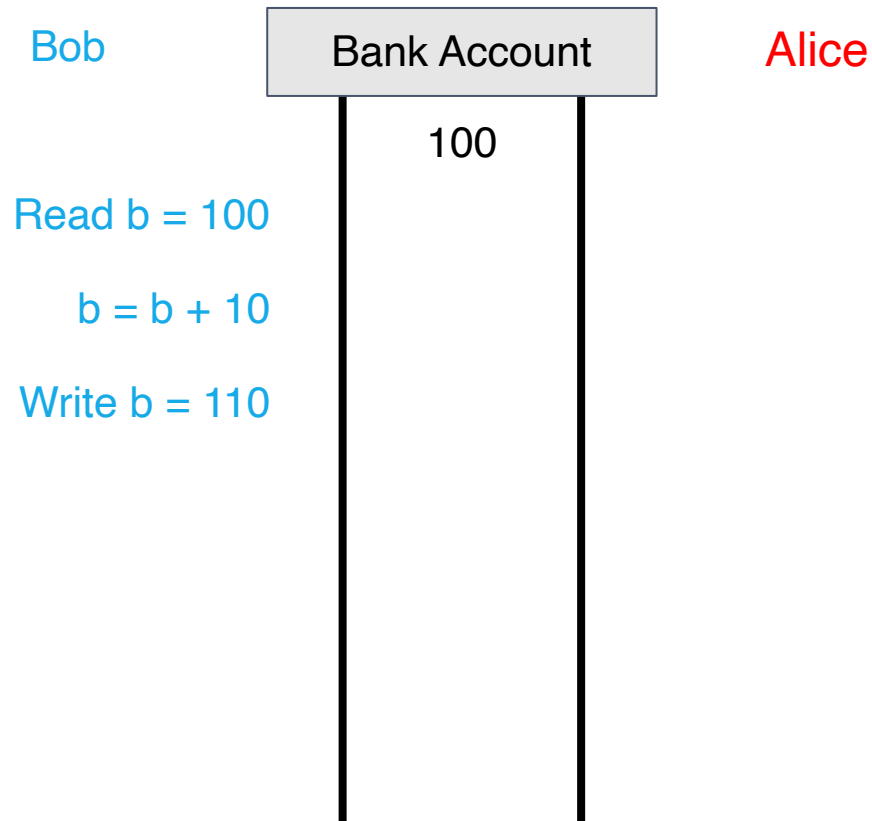
Bob



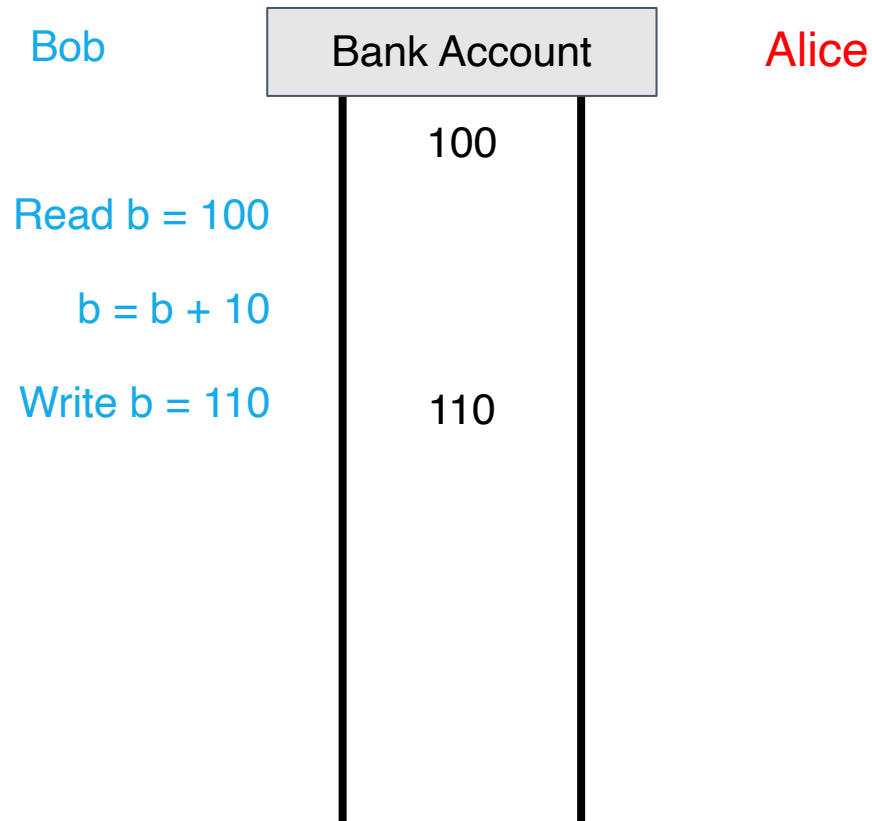
Alice

100

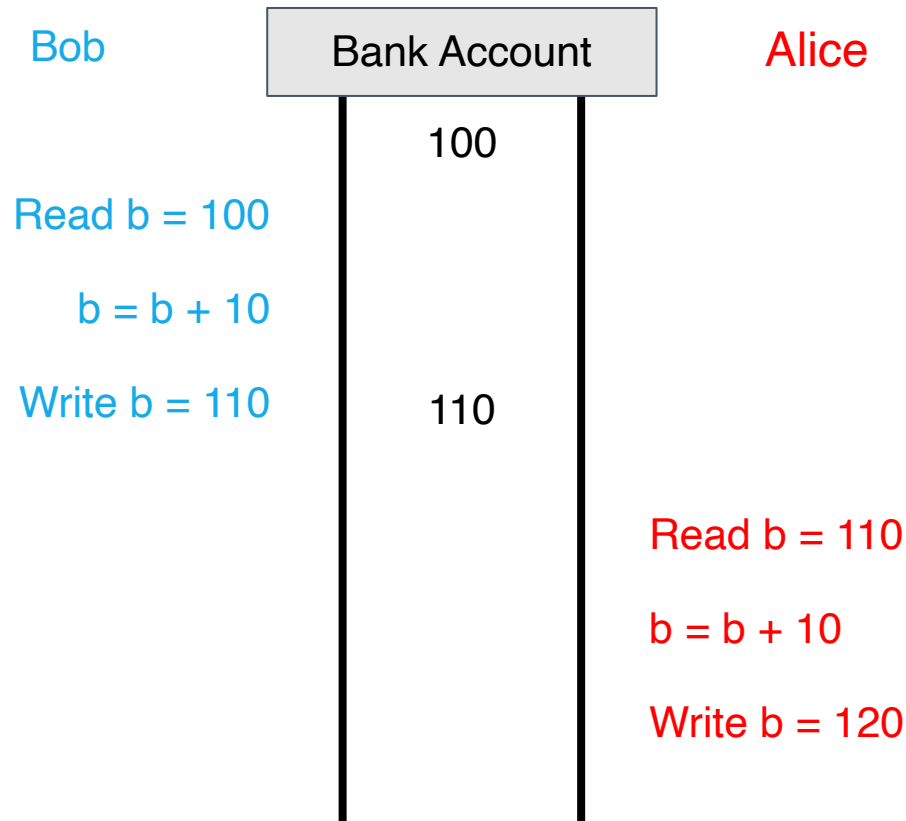
Example: Bank account



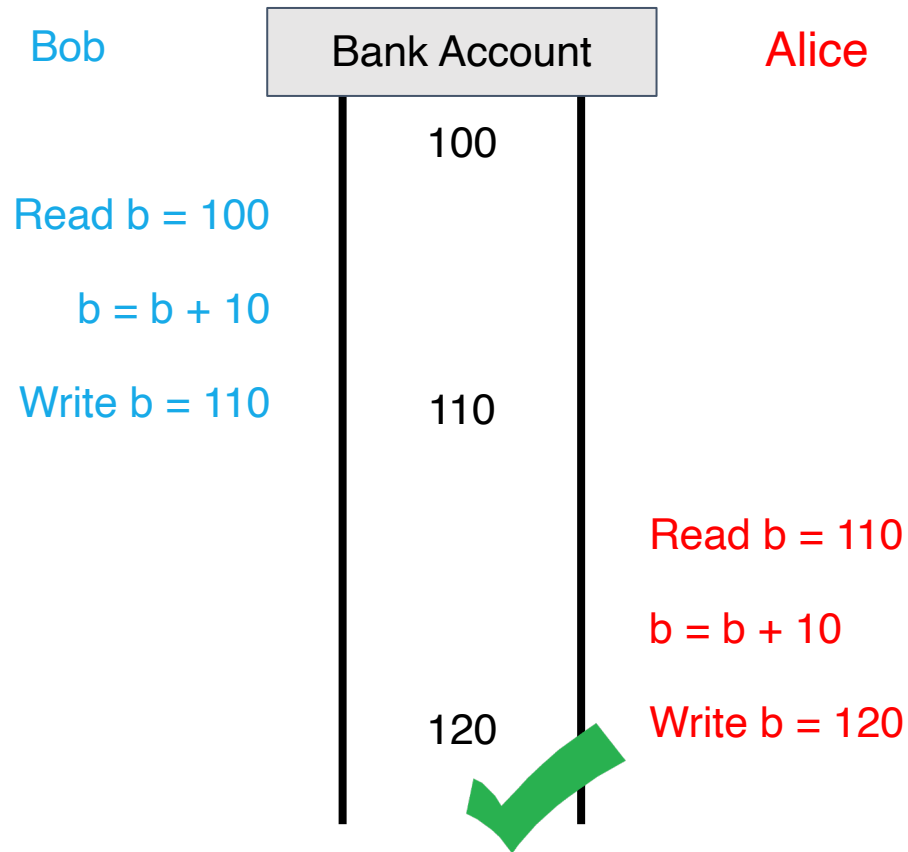
Example: Bank account



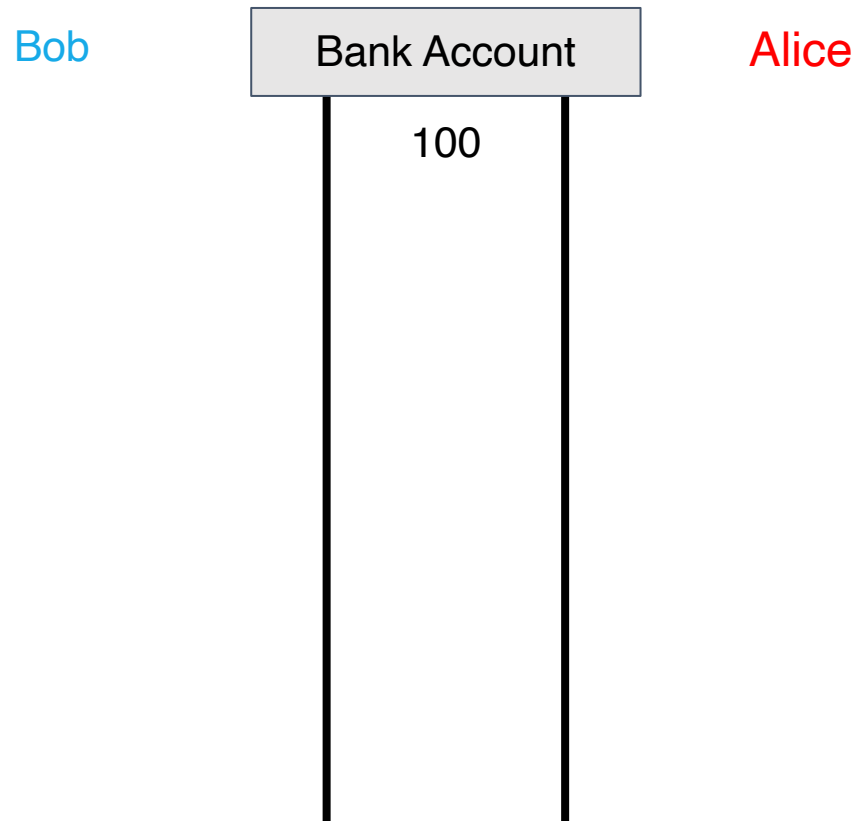
Example: Bank account



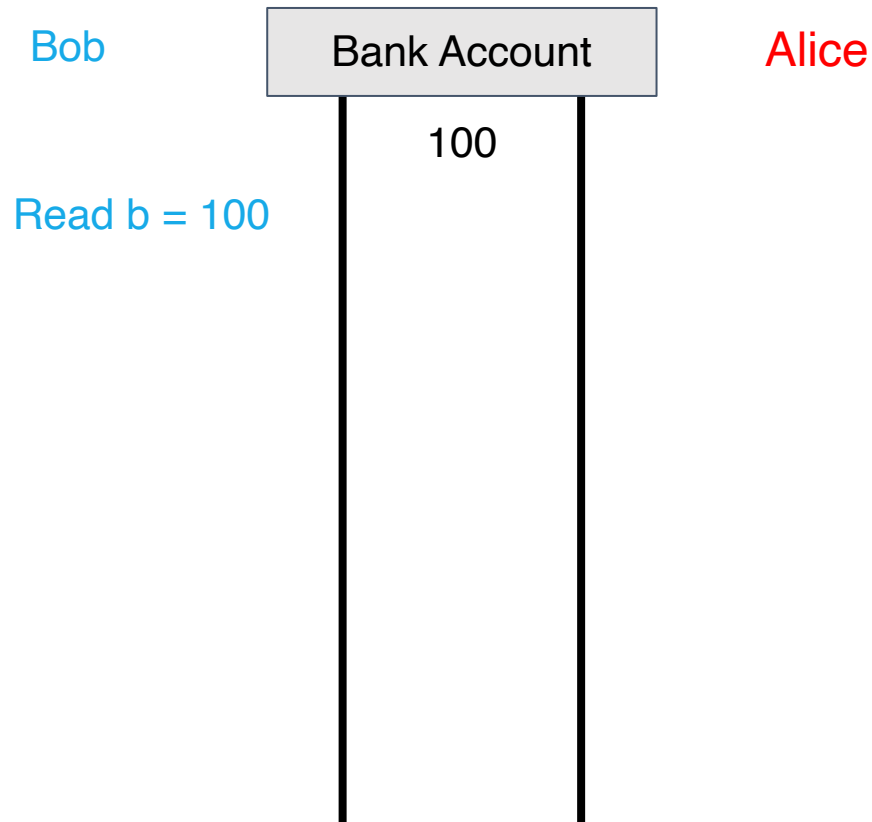
Example: Bank account



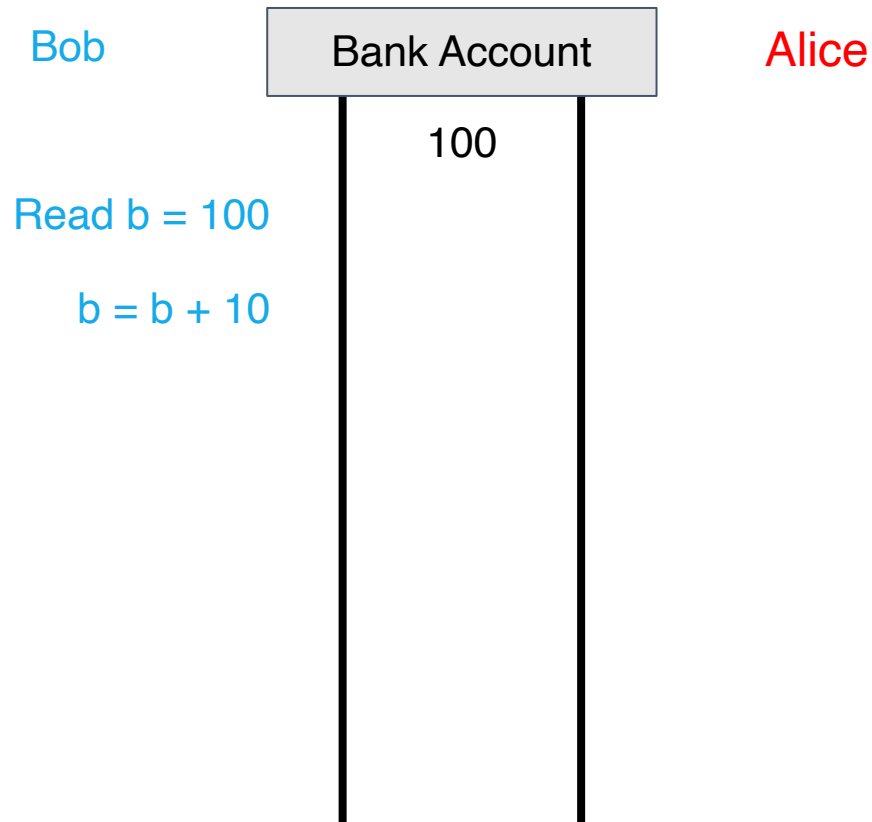
Example: Bank account



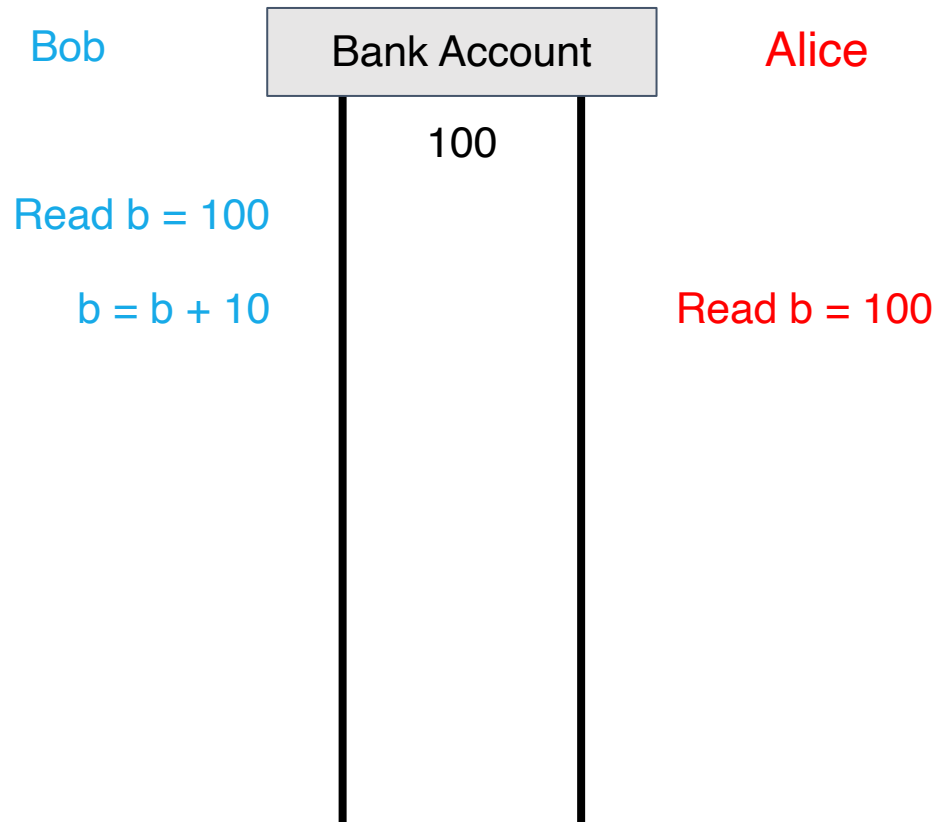
Example: Bank account



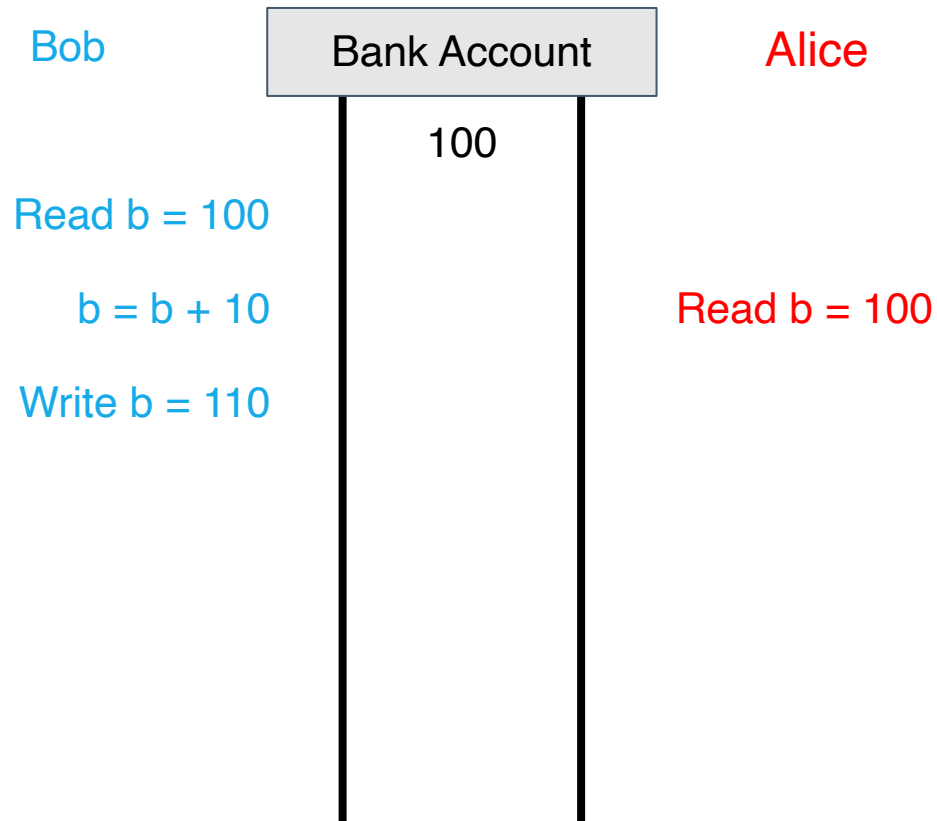
Example: Bank account



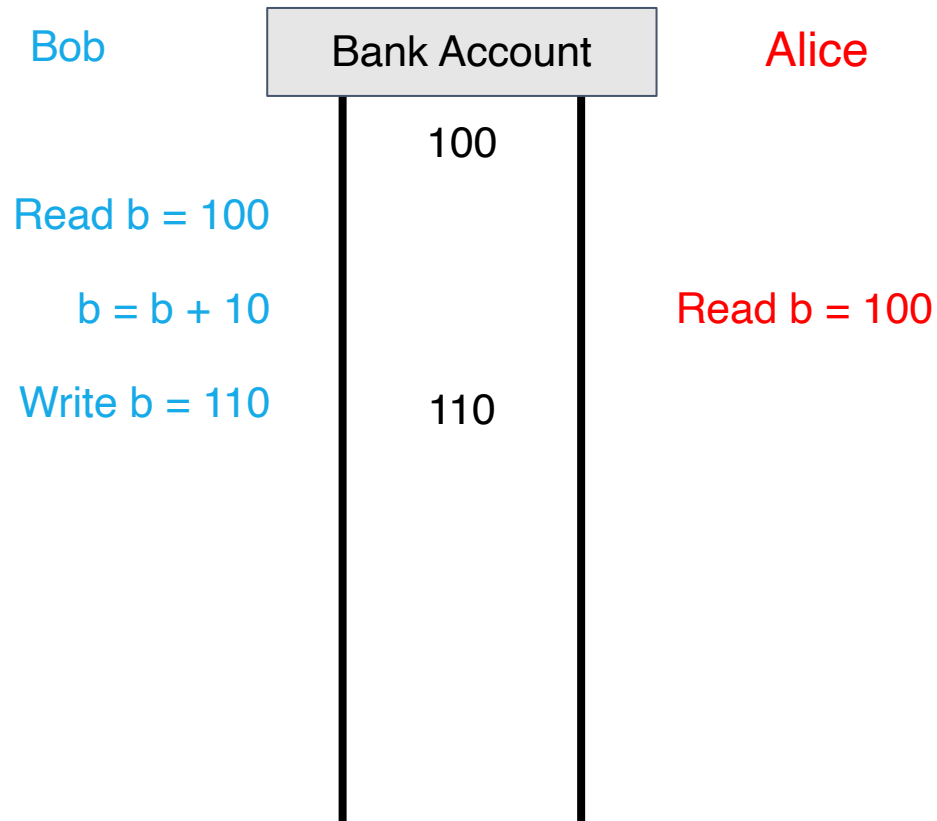
Example: Bank account



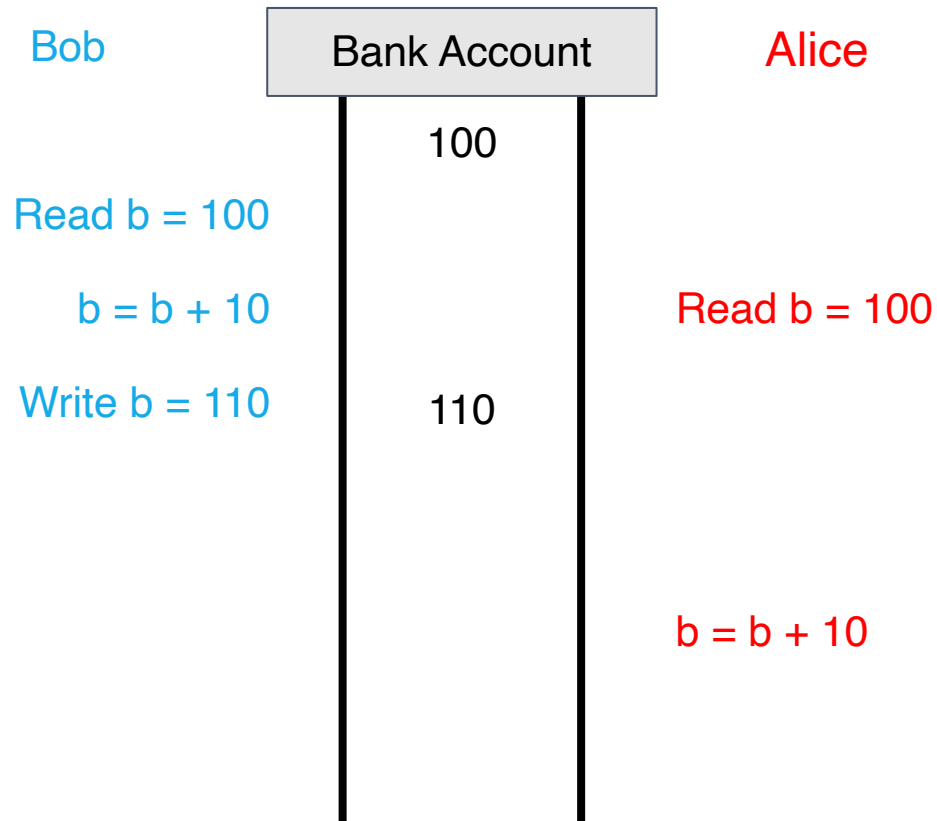
Example: Bank account



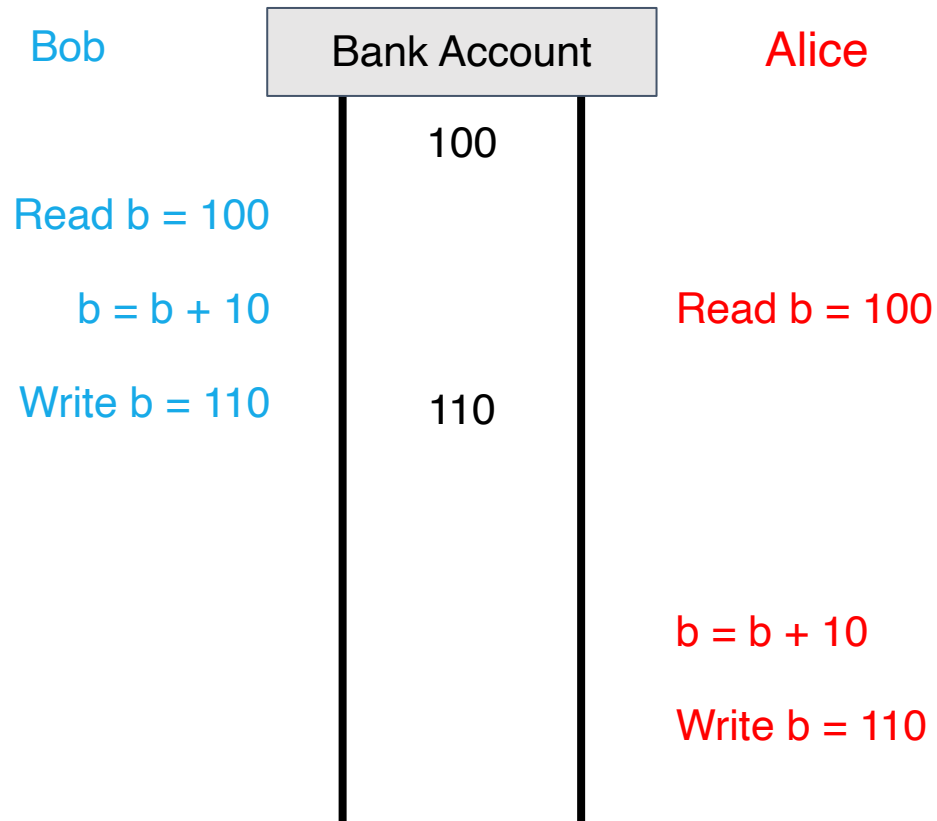
Example: Bank account



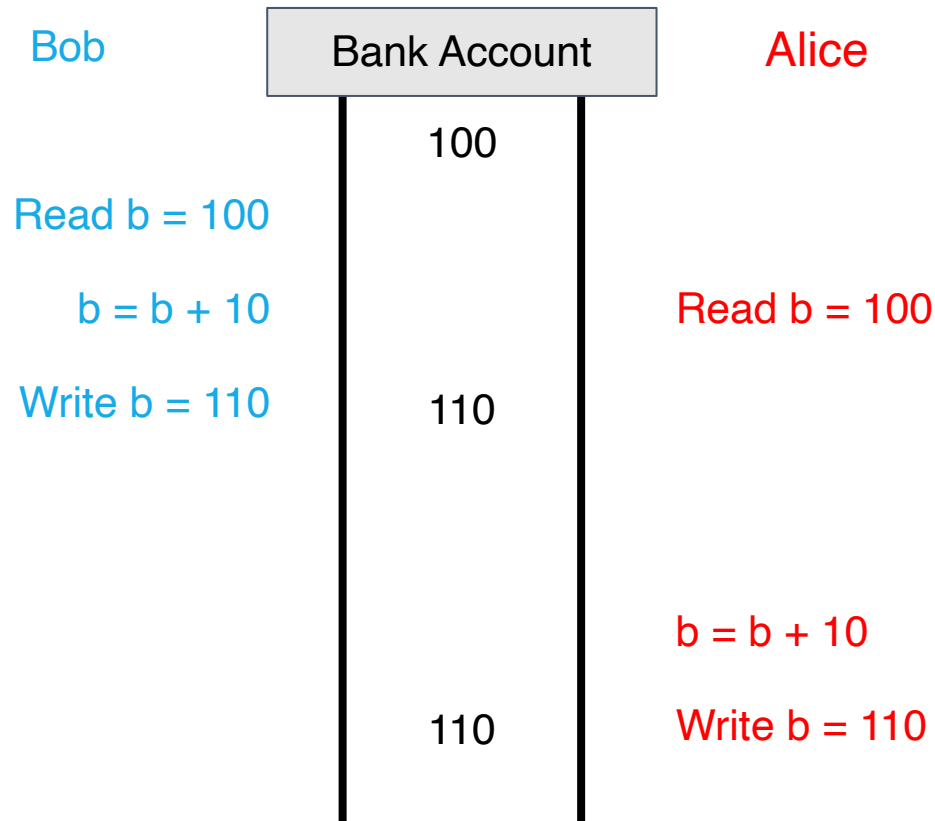
Example: Bank account



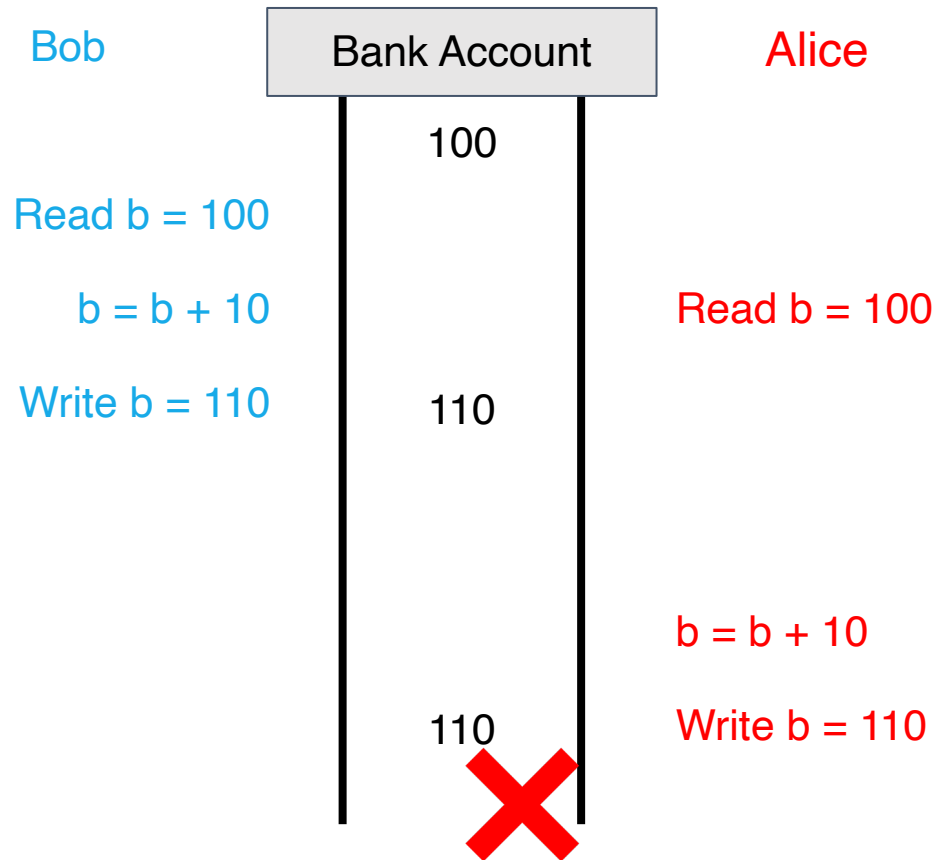
Example: Bank account



Example: Bank account



Example: Bank account



What went wrong?

- Changes to balance are not *atomic*

Root cause: Let's look at a C example...

Threaded counting example

```
1 #include <stdio.h>
2 #include "common.h"
3
4 static volatile int counter = 0;
5
6 //
7 // mythread()
8 //
9 // Simply adds 1 to counter repeatedly, in a loop
10 // No, this is not how you would add 10,000,000 to
11 // a counter, but it shows the problem nicely.
12 //
13 void *mythread(void *arg)
14 {
15     printf("%s: begin\n", (char *) arg);
16     int i;
17     for (i = 0; i < 1e7; i++) {
18         counter = counter + 1;
19     }
20     printf("%s: done\n", (char*) arg);
21     return NULL;
22 }
23
24 //
25 // main()
26 //
27 // Just launches two threads (pthread_create)
28 // and then waits for them (pthread_join)
29 //
30 int main(int argc, char *argv[])
31 {
32     pthread_t p1, p2;
33     printf("main: begin (counter = %d)\n", counter);
34     Pthread_create(&p1, NULL, mythread, "A");
35     Pthread_create(&p2, NULL, mythread, "B");
36
37     // join waits for the threads to finish
38     Pthread_join(p1, NULL);
39     Pthread_join(p2, NULL);
40     printf("main: done with both (counter = %d)\n", counter);
41     return 0;
42 }
```



```
$ git clone https://github.com/tddg/demo-ostep-code
$ cd demo-ostep-code/threads-intro
$ make
$ ./t1 <loop_count>
```

Try it yourself

Back-to-back runs

Run 1...

main: begin (counter = 0)

A: begin

B: begin

A: done

B: done

main: done with both (counter = 10706438)

Run 2...

main: begin (counter = 0)

A: begin

B: begin

A: done

B: done

main: done with both (counter = 11852529)

What exactly happened??

What exactly happened??

```
% otool -t -v thread_rc           [Mac OS X]
% objdump -d thread_rc          [Linux]
```

```
...
00000000100000d52    movl 0x2f8e, %eax
00000000100000d58    addl $0x1, %eax
00000000100000d5b    movl %eax, 0x2f8e
...
```

`counter = counter + 1;`

Concurrent access to the same memory address

OS

Thread 1

Thread 2

Value



Enter into critical section

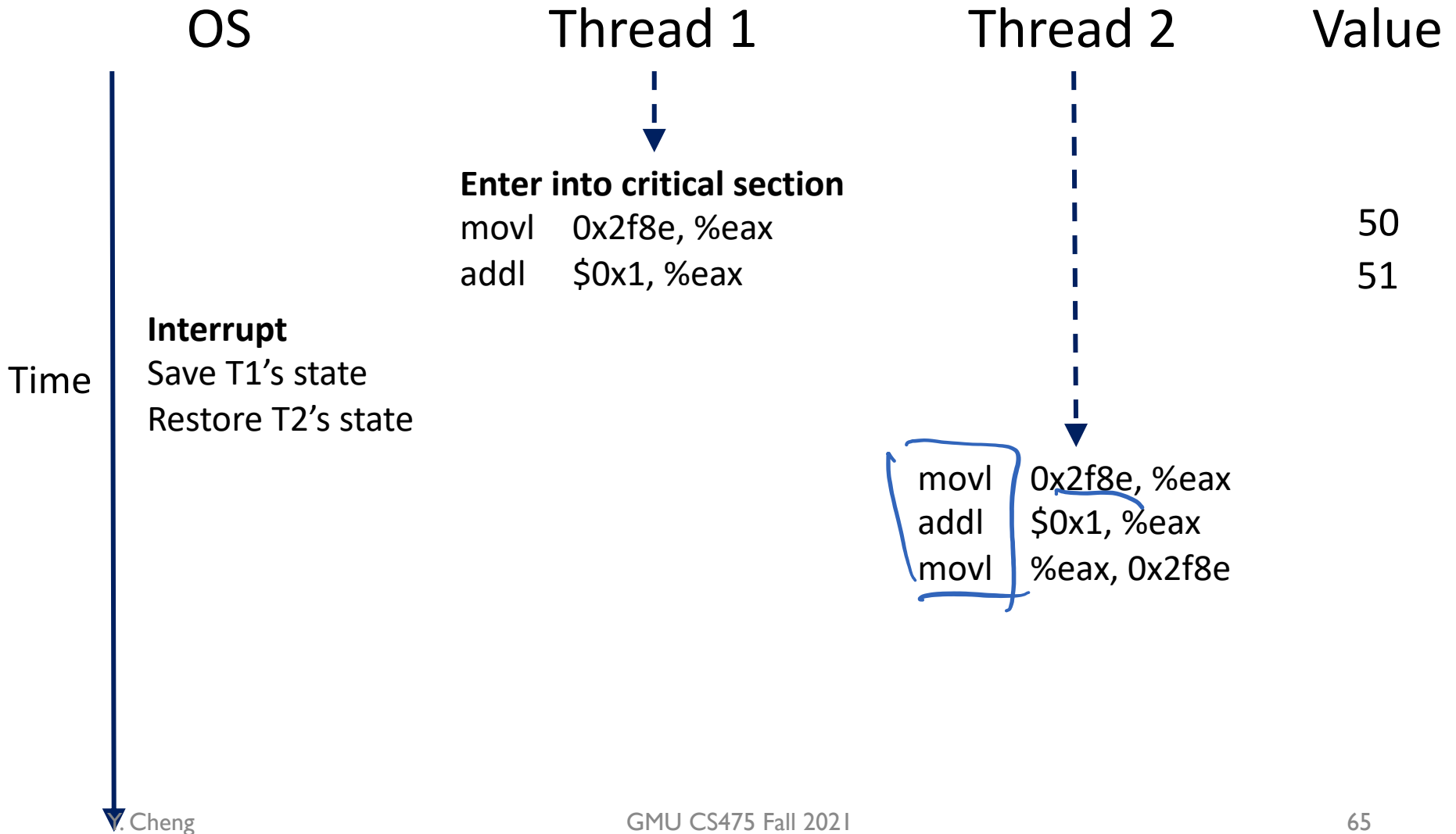
```
movl 0x2f8e, %eax  
addl $0x1, %eax
```

50

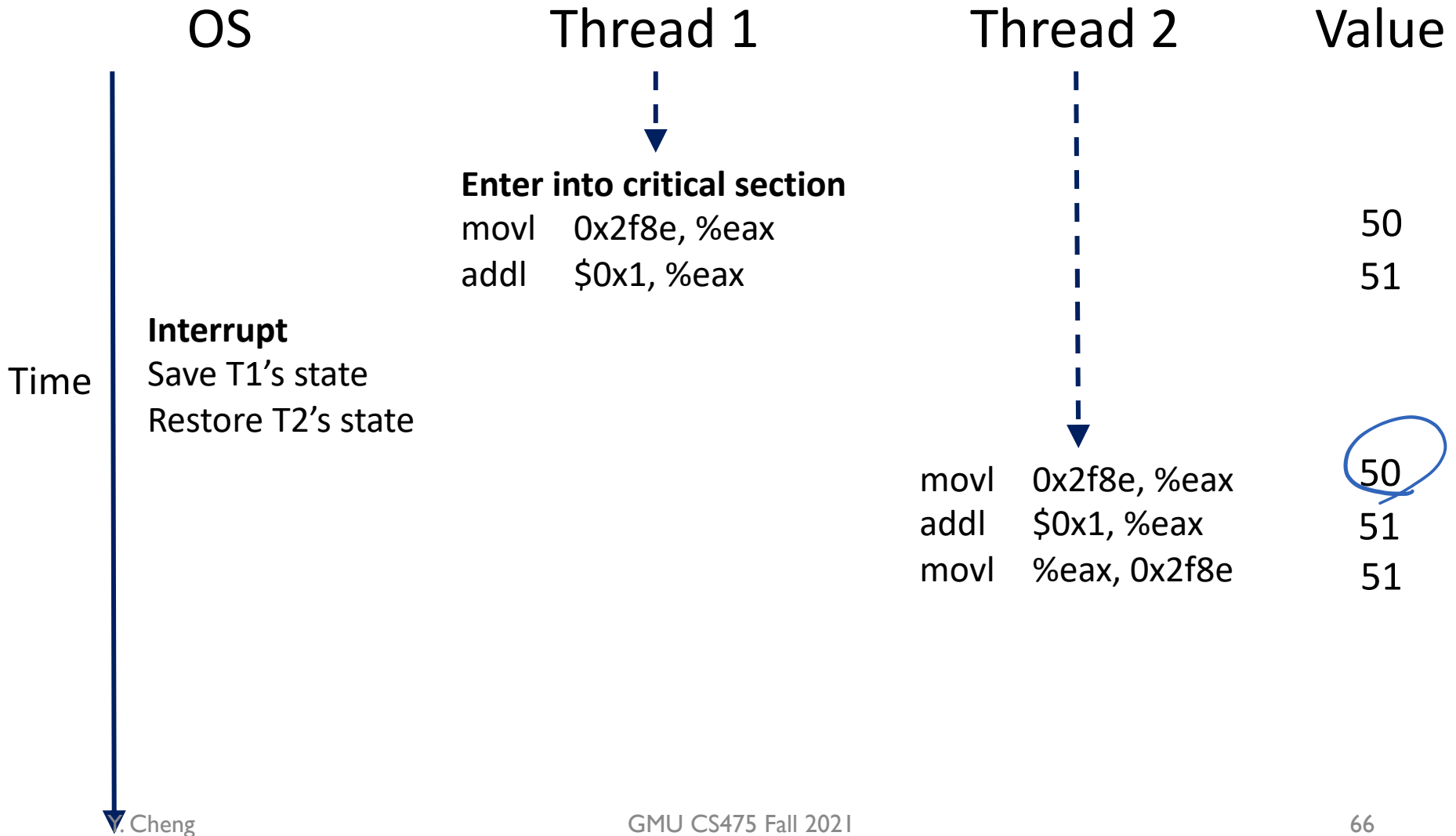
51

Time

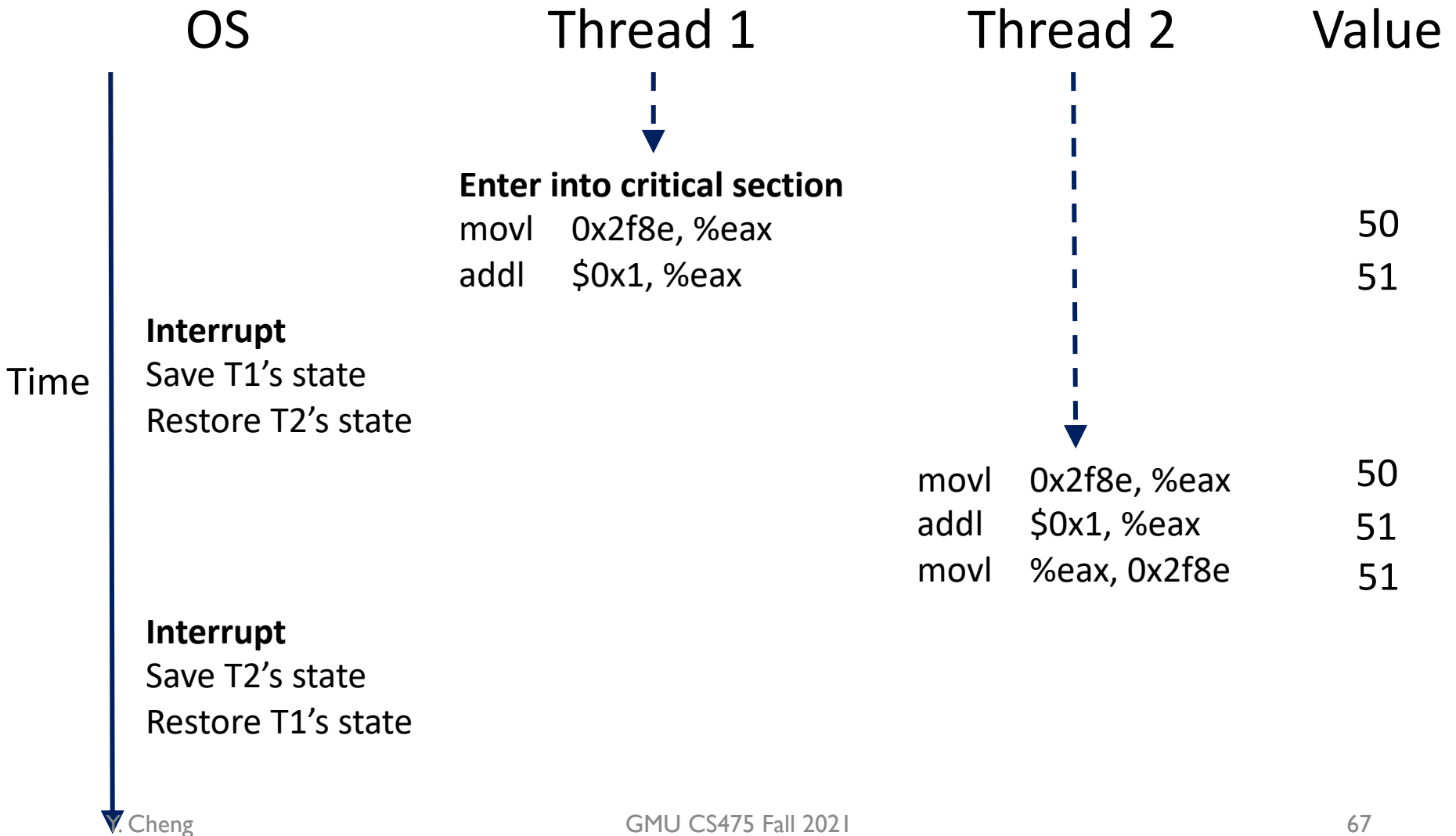
Concurrent access to the same memory address



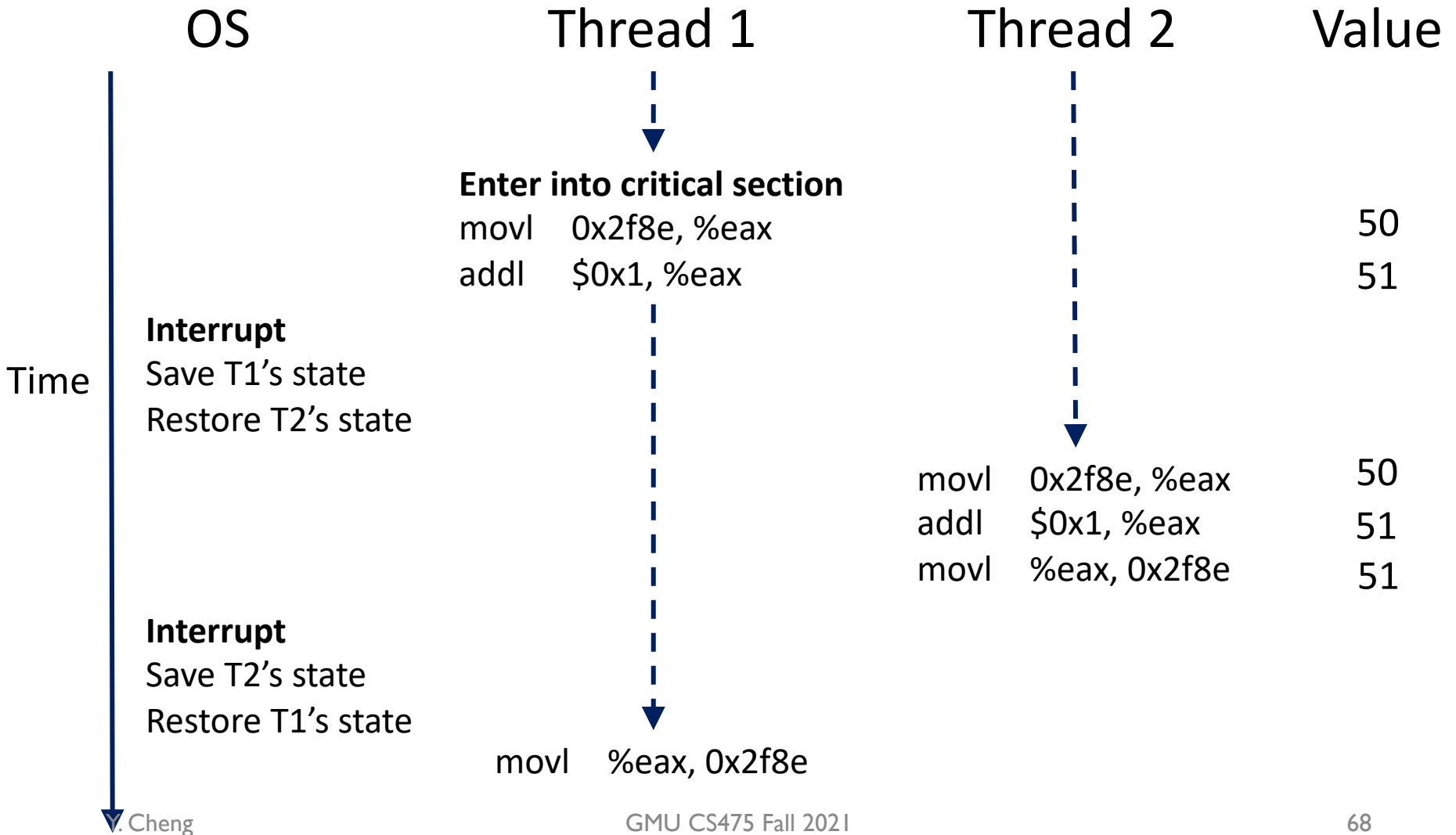
Concurrent access to the same memory address



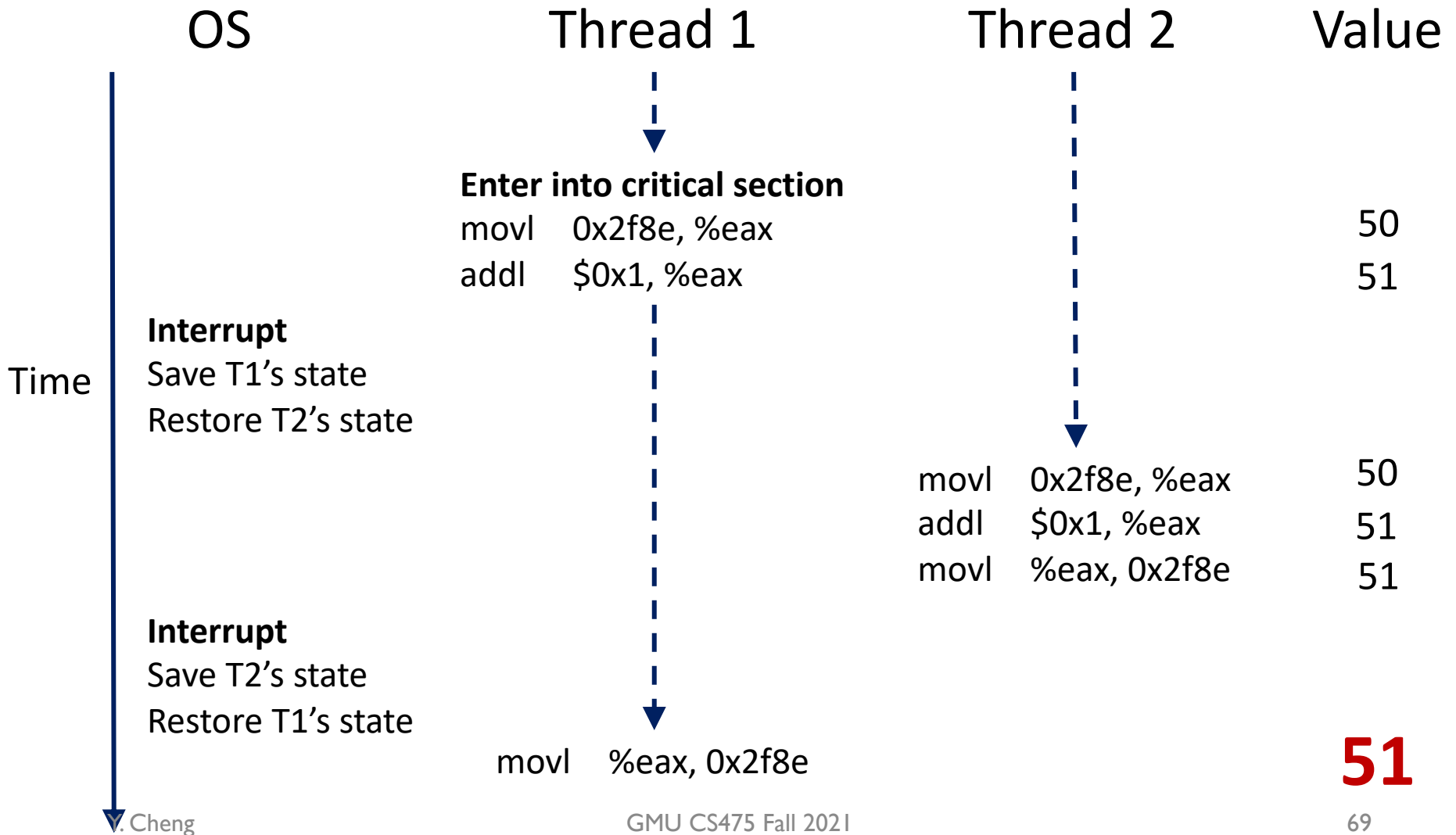
Concurrent access to the same memory address



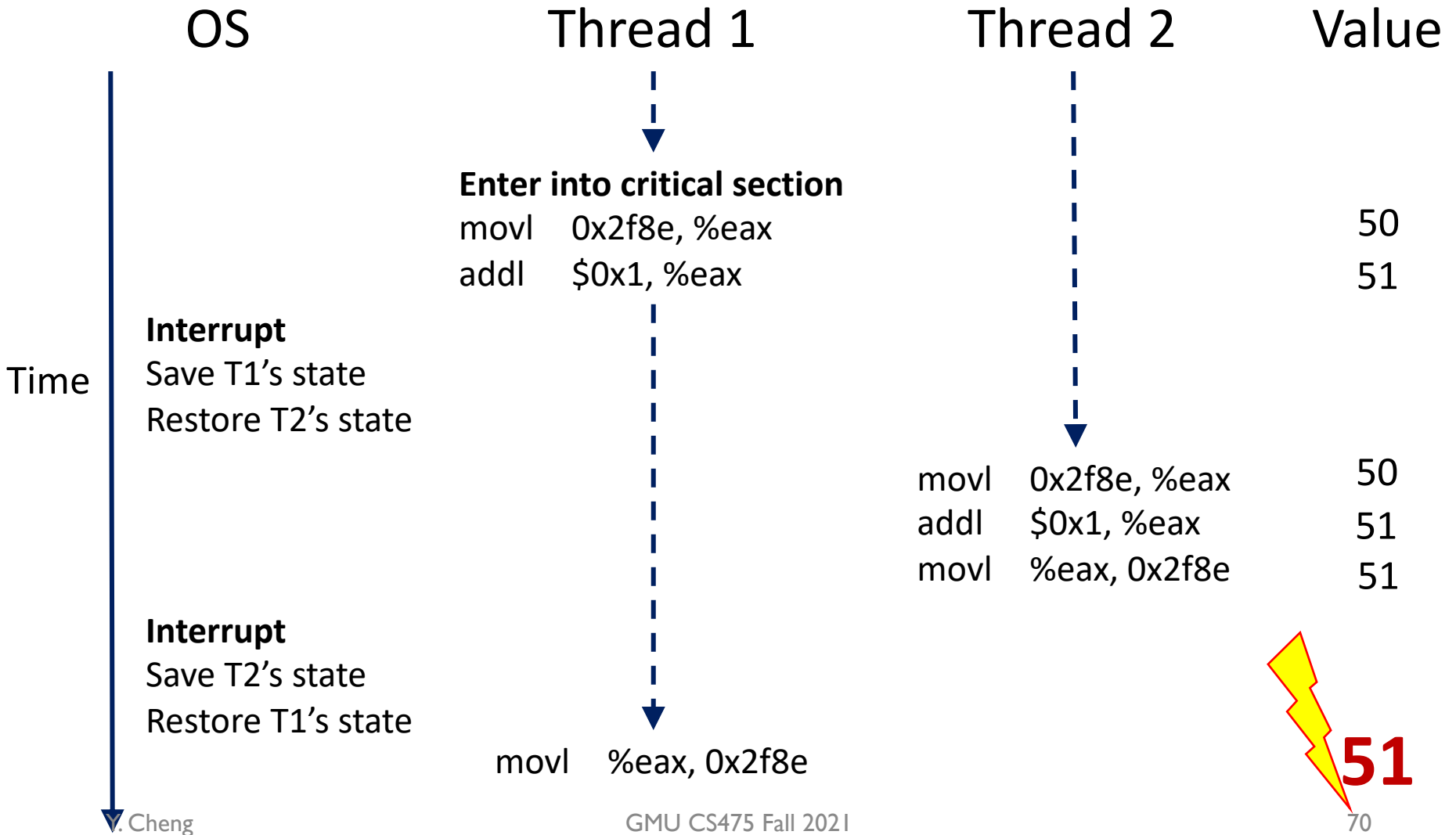
Concurrent access to the same memory address



Concurrent access to the same memory address



Concurrent access to the same memory address



Race conditions

- Observe: In a **time-shared** system, **the exact instruction execution order** cannot be predicted
 - Deterministic vs. **Non-deterministic**
- Any possible orders can happen, which result in different output across runs

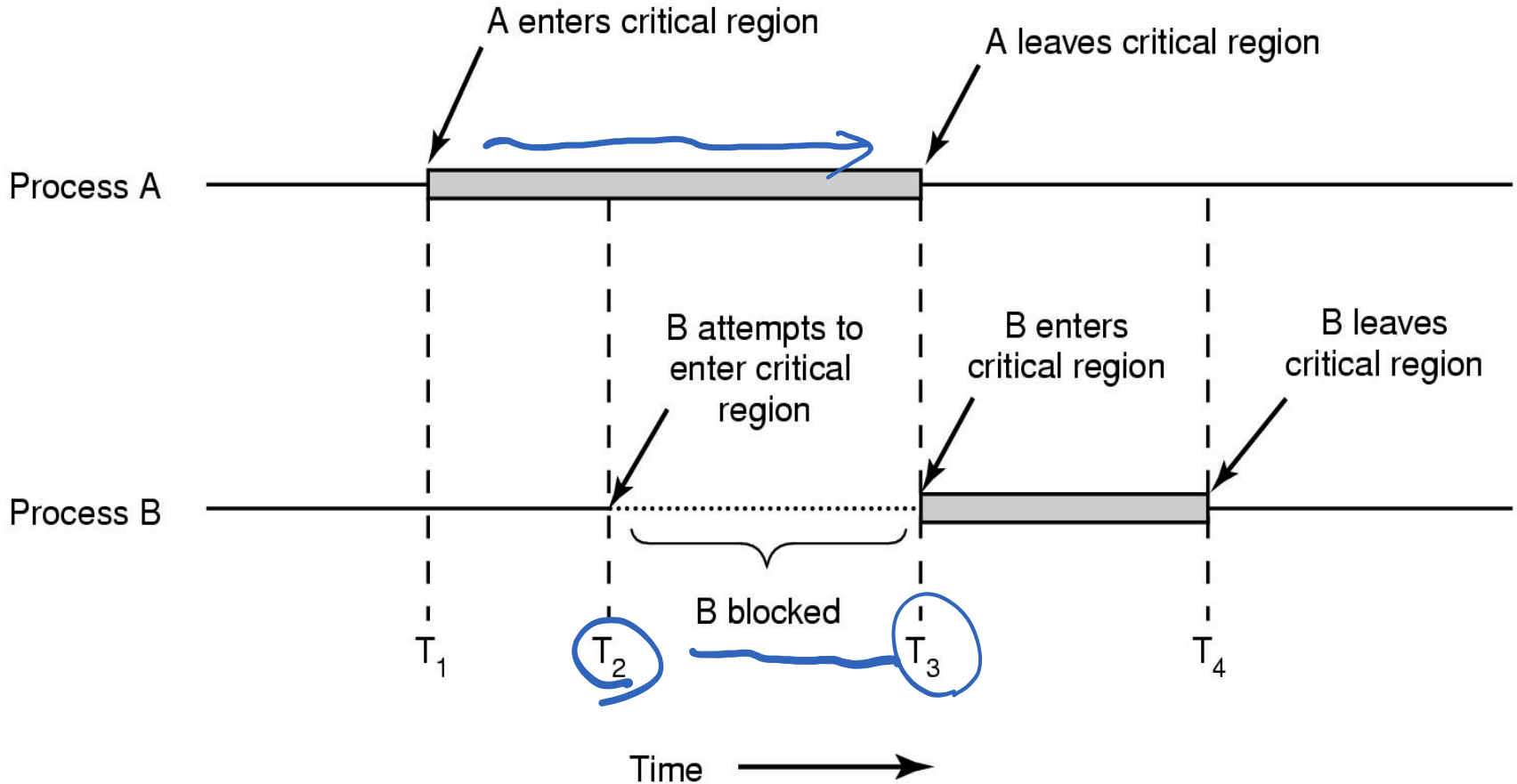
Race conditions

- Situations like this, where multiple threads are writing or reading some shared data and the final **result depends on who runs precisely when**, are called **race conditions**
 - A serious problem for any concurrent system using shared variables
- Programmers must make sure that some **high-level** code sections are executed **atomically**
 - Atomic operation: It completes in its **entirety without worrying about interruption by any other potentially conflict-causing thread**

The critical-section problem

- N threads all competing to access the shared data
- Each process/thread has a code segment, called **critical section (critical region)**, in which the shared data is accessed
- Problem – ensure that when one thread is executing in its critical section, no other thread is allowed to execute in that critical section
- The execution of the critical sections by the threads must be **mutually exclusive** in time

Mutual exclusion



Solving critical-section problem

Any solution to the problem must satisfy **four conditions!**

Mutual Exclusion:

No two threads may be simultaneously inside the same critical section

Bounded Waiting:

No thread should have to wait forever to enter a critical section

Progress:

No thread executing a code segment unrelated to a given critical section can block another thread trying to enter the same critical section

Arbitrary Speed:

No assumption can be made about the relative speed of different threads (though all threads have a non-zero speed)

Locks

- A lock is a **variable**
- Two states
 - Available or free
 - Locked or held

- • **lock()**: tries to acquire the lock
- **unlock()**: releases the lock that has been acquired by caller

Using Lock to protect shared data

- Changes to `balance` are not *atomic*

Thread A:

```
balance = balance + amount
```

Thread B:

```
balance = balance + amount
```

```
func Deposit(amount) {  
    balanceLock.lock() ←  
    read balance  
    balance = balance + amount  
    write balance ←  
    balanceLock.unlock() ←  
}
```

Using Lock to protect shared data

- Changes to `balance` are not *atomic*

Thread A:

```
balance = balance + amount
```

Thread B:

```
balance = balance + amount
```

```
func Deposit(amount) {  
    balanceLock.lock()  
    read balance  
    balance = balance + amount  
    write balance  
    balanceLock.unlock()  
}
```

***Critical
section***

Concurrency

- Process vs. thread
- Race conditions
- Locks
- Concurrency in Go
 - Two synchronization mechanisms
 - Locks
 - Channels

Two synchronization mechanisms in Go

- **Locks:** limit access to a critical section
 - Access to a critical section (e.g., shared variables) must be mutually exclusive
- **Channels:** pass information across threads using a queue

Mutex locks in Go

```
package account

import "sync"

type Account struct {
    balance int
    lock sync.Mutex
}
```

Mutex locks in Go

```
package account

import "sync"

type Account struct {
    balance int
    lock sync.Mutex
}

func NewAccount(init int) Account
    return Account{balance: init}
}
```

Mutex locks in Go

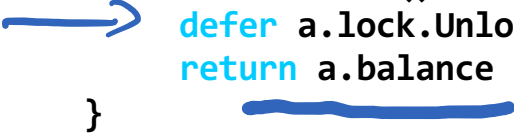
```
package account
```

```
import "sync"
```

```
type Account struct {  
    balance int  
    lock sync.Mutex  
}
```

```
func NewAccount(init int) Account  
    return Account{balance: init}  
}
```

```
func (a *Account) CheckBalance() int {  
    a.lock.Lock()  
    defer a.lock.Unlock()  
    return a.balance  
}
```



Mutex locks in Go

```
package account

import "sync"

type Account struct {
    balance int
    lock sync.Mutex
}

func NewAccount(init int) Account
    return Account{balance: init}
}
```

```
func (a *Account) CheckBalance() int {
    a.lock.Lock()
    defer a.lock.Unlock()
    return a.balance
}
```

```
func (a *Account) Withdraw(v int) {
    a.lock.Lock()
    defer a.lock.Unlock()
    a.balance -= v
}
```

```
func (a *Account) Deposit(v int) {
    a.lock.Lock()
    defer a.lock.Unlock()
    a.balance += v
}
```

Read write locks in Go

```
package account

import "sync"

type Account struct {
    balance int
    lock sync.RWMutex
}

func NewAccount(init int) Account
    return Account{balance: init}
}
```

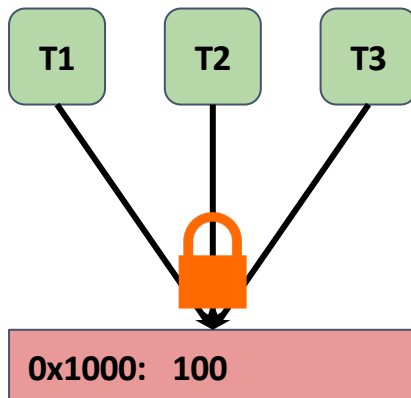
```
func (a *Account) CheckBalance() int {
    a.lock.RLock()
    defer a.lock.RUnlock()
    return a.balance
}
```

```
func (a *Account) Withdraw(v int) {
    a.lock.Lock()
    defer a.lock.Unlock()
    a.balance -= v
}
```

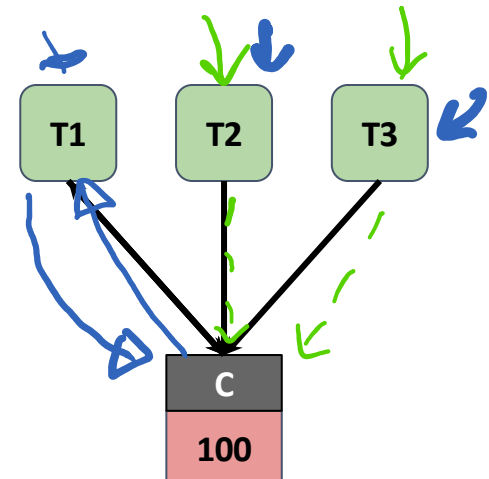
```
func (a *Account) Deposit(v int) {
    a.lock.Lock()
    defer a.lock.Unlock()
    a.balance += v
}
```

Two solutions to the same problem

- Locks:
 - Multiple threads can reference same memory location
 - Use lock to ensure only one thread is updating it at any time



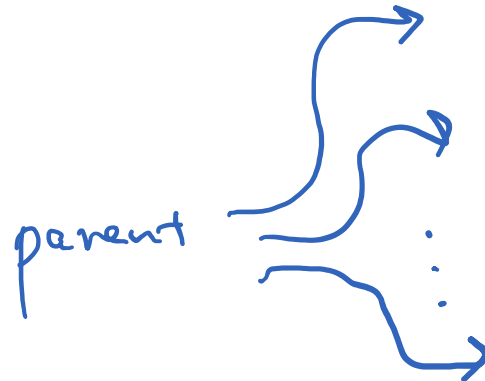
- Channels:
 - Data item initially stored in channel
 - Threads must request item from channel, make updates, and return item to channel



Go channels

```
// Launch workers
for i := 0; i < numWorkers; i++ {
    go func() {
        // ... do some work
    }()
}
```

- In Go, *channels* and *goroutines* are more idiomatic than locks



Go channels

buffered channel.

```
result := make(chan int, numWorkers)
```

```
// Launch workers
```

```
for i := 0; i < numWorkers; i++ {  
    go func() {  
        // ... do some work  
        result <- i  
    }()  
}
```

- In Go, *channels* and *goroutines* are more idiomatic than locks

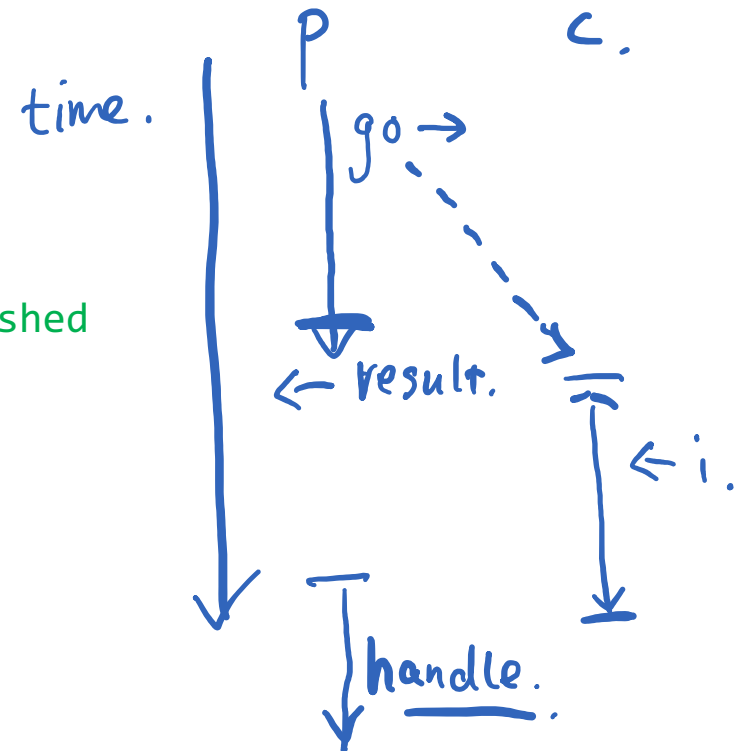
Go channels

```
result := make(chan int, numWorkers)
```

```
// Launch workers  
for i := 0; i < numWorkers; i++ {  
    → go func() {  
        // ... do some work  
        result <- i  
    }()  
}
```

```
↪ // Wait until all worker threads have finished  
for i := 0; i < numWorkers; i++ {  
    handleResult(<-result)  
}  
fmt.Println("Done!")
```

- In Go, *channels* and *goroutines* are more idiomatic than locks



Bank account code (using channels)

```
package account

type Account struct {
    // Fill in Here
}

func NewAccount(init int) Account {
    // Fill in Here
}

func (a *Account) CheckBalance() int {
    // What goes Here?
}

func (a *Account) Withdraw(v int) {
    // ???
}


func (a *Account) Deposit(v int) {
    // ???
}
```

Bank account code (using channels)

```
package account
```

```
type Account struct {  
    balance chan int  
}
```

```
func NewAccount(init int) Account {  
    a := Account make(chan int, 1)  
    a.balance <- init  
    return a  
}
```




```
func (a *Account) CheckBalance() int {  
    bal := <-a.balance  
    a.balance <- bal  
    return bal  
}
```

```
func (a *Account) Withdraw(v int) {  
    bal := <-a.balance  
    a.balance <- (bal - v)  
}
```

```
func (a *Account) Deposit(v int) {  
    bal := <-a.balance  
    a.balance <- (bal + v)  
}
```

select statement in Go

- `select` allows a goroutine to wait on multiple channels at once




```
for {  
    select {  
        case money := <-dad:  
            buySnacks(money)  
        case money := <-mom:  
            buySnacks(money)  
        case default:  
            starve()  
            time.Sleep(5 * time.Second)  
    }  
}
```

Handle timeouts using select

```
result := make(chan int)
timeout := make(chan bool)

// Asynchronously request an
// answer from server, timing
// out after X seconds
askServer(result, timeout)

// Wait on both channels
select {
    case res := <-result:
        handleResult(res)
    case <-timeout:
        fmt.Println("Timeout!")
}
```



```
func askServer(
    result chan int,
    timeout chan bool) {

    // Start timer
    go func() {
        time.Sleep(5 * time.Second)
        timeout <- true
    }()

    // Ask server
    go func() {
        response := // ... send RPC
        result <- response
    }()
}
```

Handle timeouts using select

```
result := make(chan int)
timeout := make(chan bool)

// Asynchronously request an
// answer from server, timing
// out after X seconds
askServer(result, timeout)

// Wait on both channels
select {
  → case res := <-result:
      handleResult(res)
  → case <-timeout:
      fmt.Println("Timeout!")
}
```

```
func askServer(
    result chan int,
    timeout chan bool) {

    // Start timer
    go func() {
        time.Sleep(5 * time.Second)
        timeout <- true
    }()

    // Ask server
    go func() {
        response := // ... send RPC
        result <- response
    }()
}
```