CS202 (003): Operating Systems Concurrency

Instructor: Jocelyn Chen

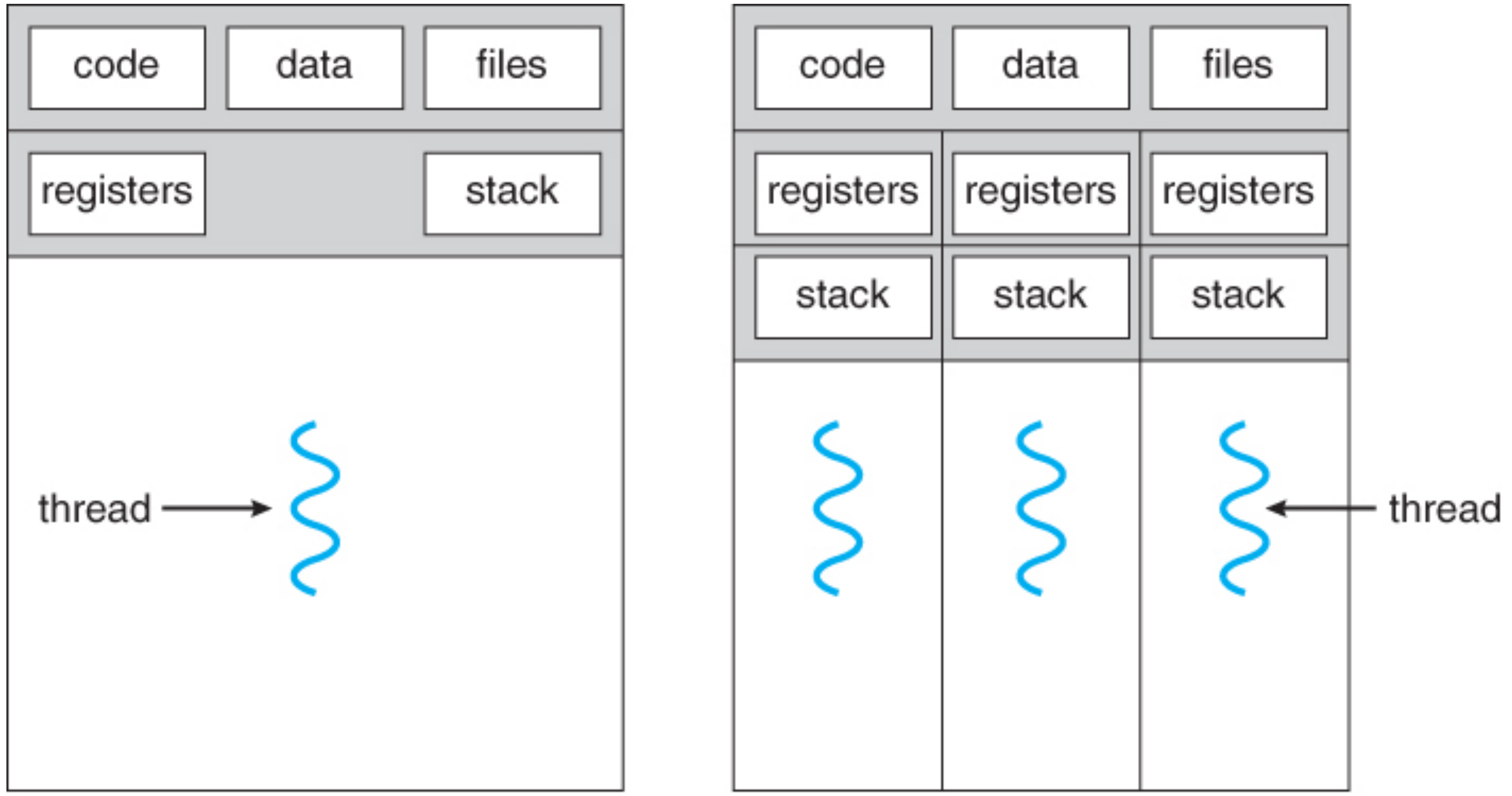
Most of the materials covered in this slide come from the lecture notes of Mike Walfish's CS202

Last time…

Threading

Lightweight units of execution within a process

That means, you can do concurrent execution within a process using thread

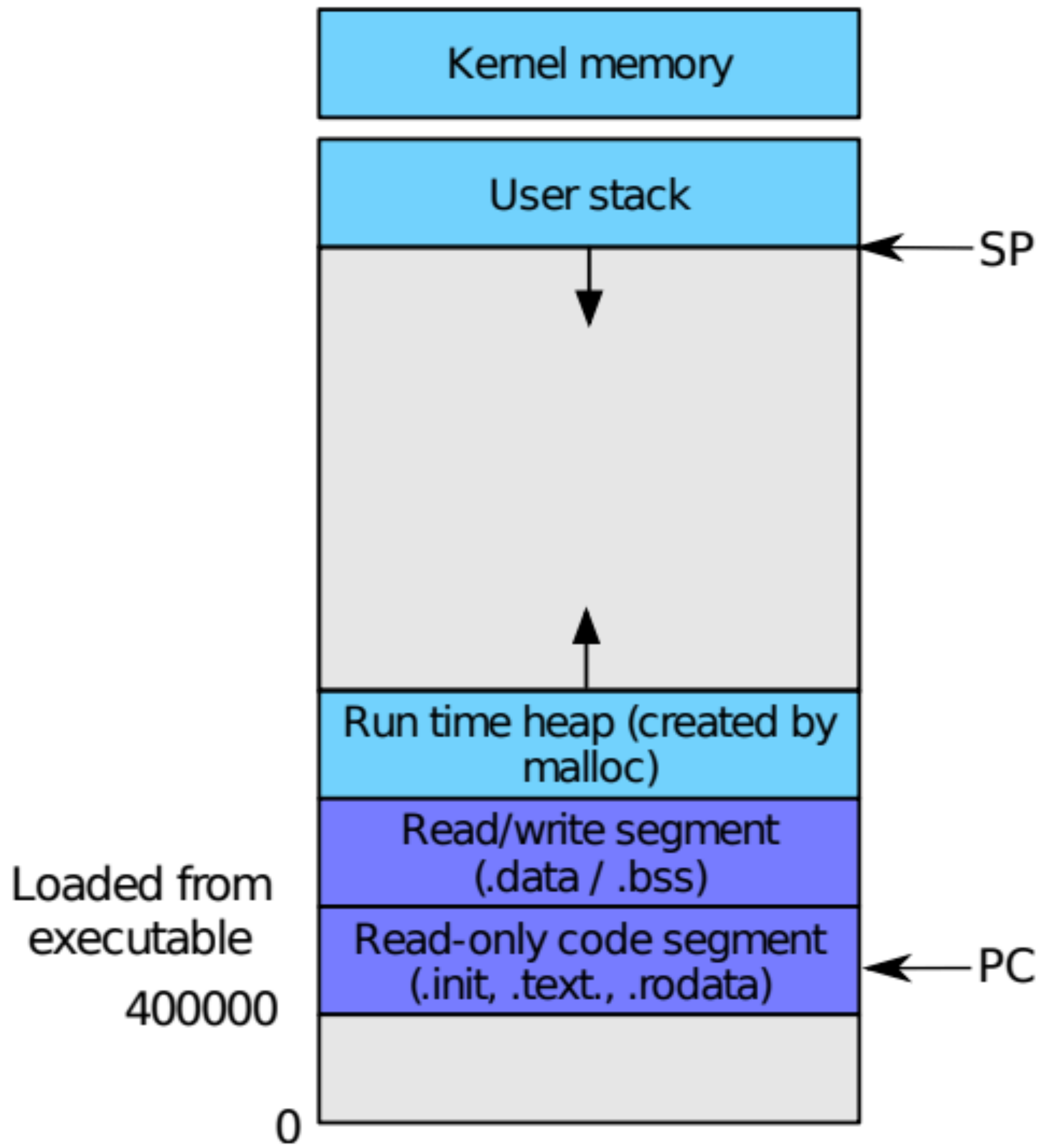


single-threaded process

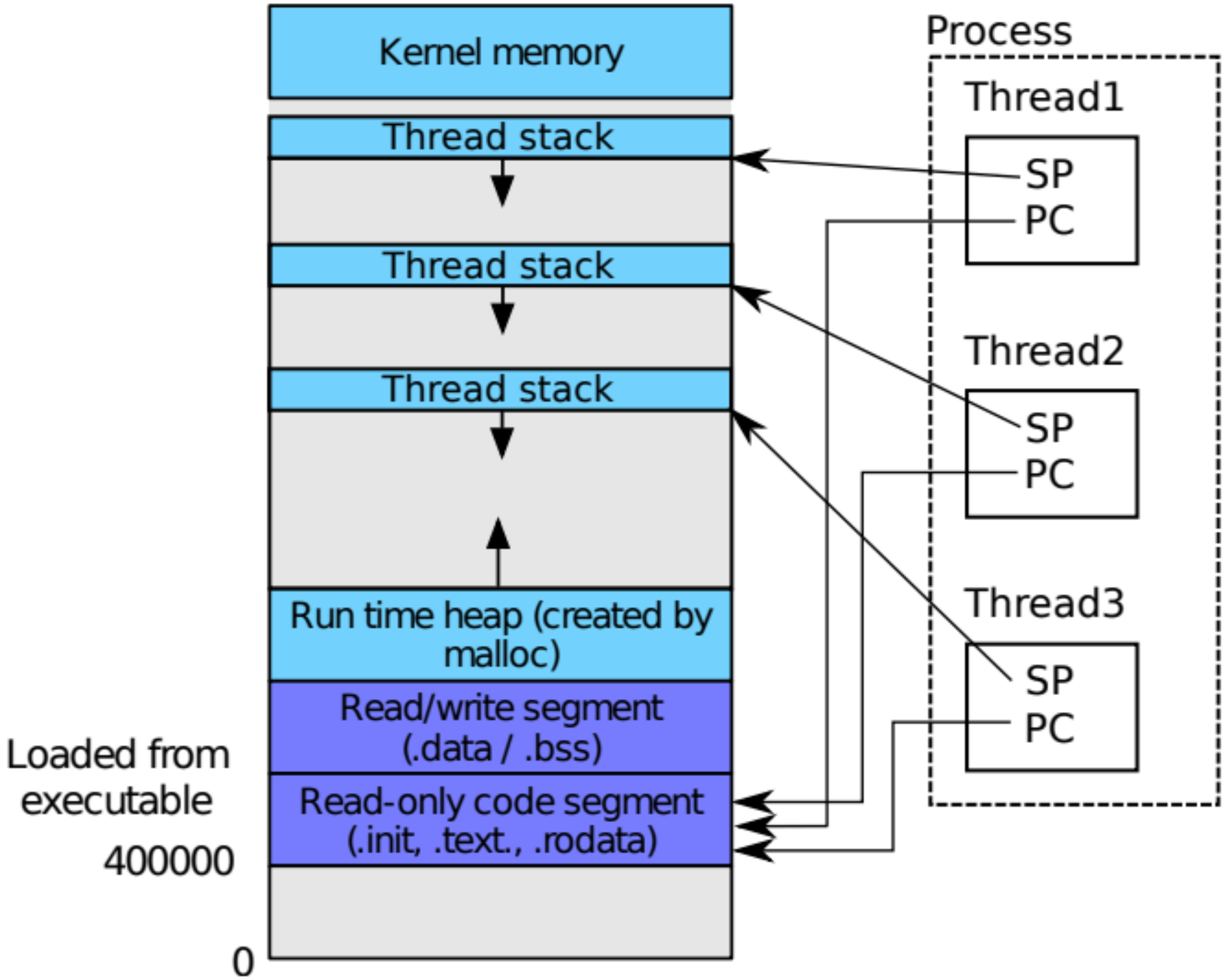
multithreaded process

https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

Process address space



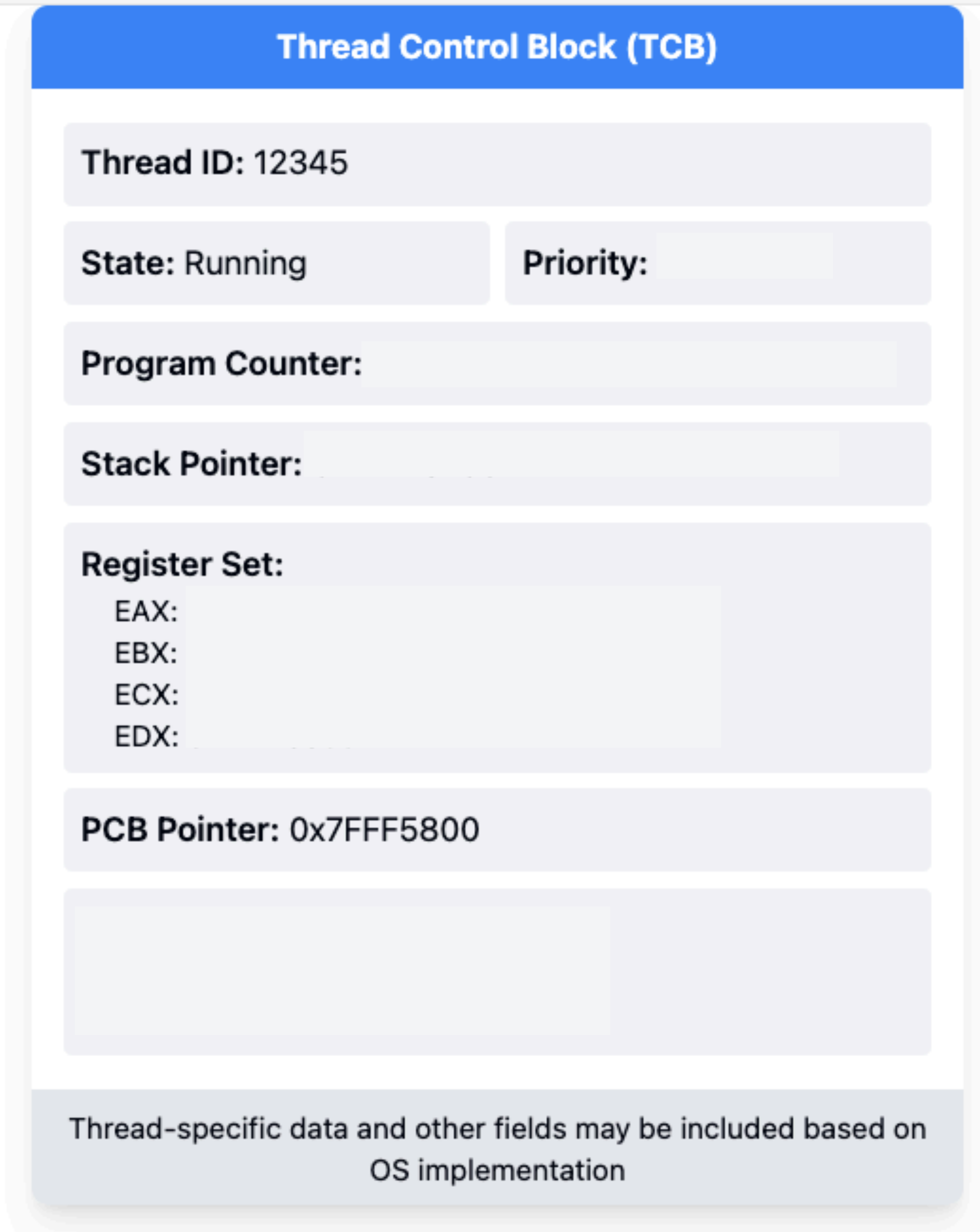
Process address space with threads



https://jhuopsys.github.io/spring2024/lectures/lecture04.pdf

TCB (thread control block)

a data structure **inside the kernel** that contains thread-specific information needed for managing the thread.



Interface to Threads

How do we create threads?

```
tid thread_create(void (*fn) (void *), void *arg);

void thread_exit();

void thread_join(tid thr);
```

And a lot more synchronization primitives

Concurrency

Simultaneous execution of multiple tasks

Broader concept than just threading

multiple CPUs and common memory

Multiple computers connected via a network

Allows CPU to work on other tasks while waiting for I/O to complete

the OS was the **first concurrent program**, and many techniques were created for use within the OS

Concurrency is **HARD**

Difficult to reason about all possible interleaving

Race conditions, deadlocks, livelocks, starvations, …

Will talk more in the following lectures.

Understanding Interleavings

```c
int x;
int main(int argc, char** argv) {
    tid tid1 = thread_create(f, NULL);
    tid tid2 = thread_create(g, NULL);

    thread_join(tid1);
    thread_join(tid2);

    printf("%d\n", x);
}

void f() {
    x = 1;
    thread_exit();
}

void g() {
    x = 2;
    thread_exit();
}
```

Understanding Interleavings

```
int x;
int y = 12;
int main(int argc, char** argv) {
    tid tid1 = thread_create(f, NULL);
    tid tid2 = thread_create(g, NULL);

    thread_join(tid1);
    thread_join(tid2);

    printf("%d\n", x);
}

void f() {
    x = y + 1;
    thread_exit();
}

void g() {
    y = y * 2;
    thread_exit();
}
```

Understanding Interleavings

```c
int x = 0;
int main(int argc, char** argv) {
    tid tid1 = thread_create(f, NULL);
    tid tid2 = thread_create(g, NULL);

    thread_join(tid1);
    thread_join(tid2);

    printf("%d\n", x);
}

void f() {
    x = x + 1;
    thread_exit();
}

void g() {
    x = x + 2;
    thread_exit();
}
```

Handout 1(c)

```
  f()
1 movq 0x5000, %rbx     # load from address 0x5000 into register
2 addq $1, %rbx         # add 1 to the register's value
3 movq %rbx, 0x5000     # store back

  g()
4 movq 0x5000, %rbx     # load from address 0x5000 into register
5 addq $2, %rbx         # add 2 to the register's value
6 movq %rbx, 0x5000     # store back
```

Understanding Interleavings

```
struct List_elem {
    int data;
    struct List_elem* next;
};

List_elem* head = 0;

insert(int data) {
    List_elem* l = new List_elem;
    l->data = data;
    l->next = head;
    head = l;
}
```

* this is pseudocode

What happens if two threads execute insert() at once and we get the following interleaving?

```
thread 1: l->next = head
thread 2: l->next = head
thread 2: head = l;
thread 1: head = l;
```

The list is broken!

Understanding Interleavings

```
void producer (void *ignored) {
    for (;;) {
        /* next line produces an item and puts it in nextProduced */
        nextProduced = means_of_production();
        while (count == BUFFER_SIZE)
            ; // do nothing
        buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
    }}

void consumer (void *ignored) {
    for (;;) {
        while (count == 0)
            ; // do nothing
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        /* next line abstractly consumes the item */
        consume_item(nextConsumed);
}}
```

* this is pseudocode

```
assuming count++ compiles to:
reg1 <- count # load
reg1 <- reg1 + 1 # increment register
count <- reg1 # store

assuming count-- compiles to:
reg2 <- count # load
reg2 <- reg2 - 1 # decrement register
count <- reg2 # store
```

What happens if we get the following interleaving?

```
reg1 <- count
reg1 <- reg1 + 1
reg2 <- count
reg2 <- reg2 - 1
count <- reg1
count <- reg2
```

The count is incorrect!

We call these situation a *race condition*

Or more specifically, a data race

It arises if multiple threads of execution enter the critical section at roughly the same time; both attempt to update the shared data structure, leading to a surprising (and perhaps undesirable) outcome.

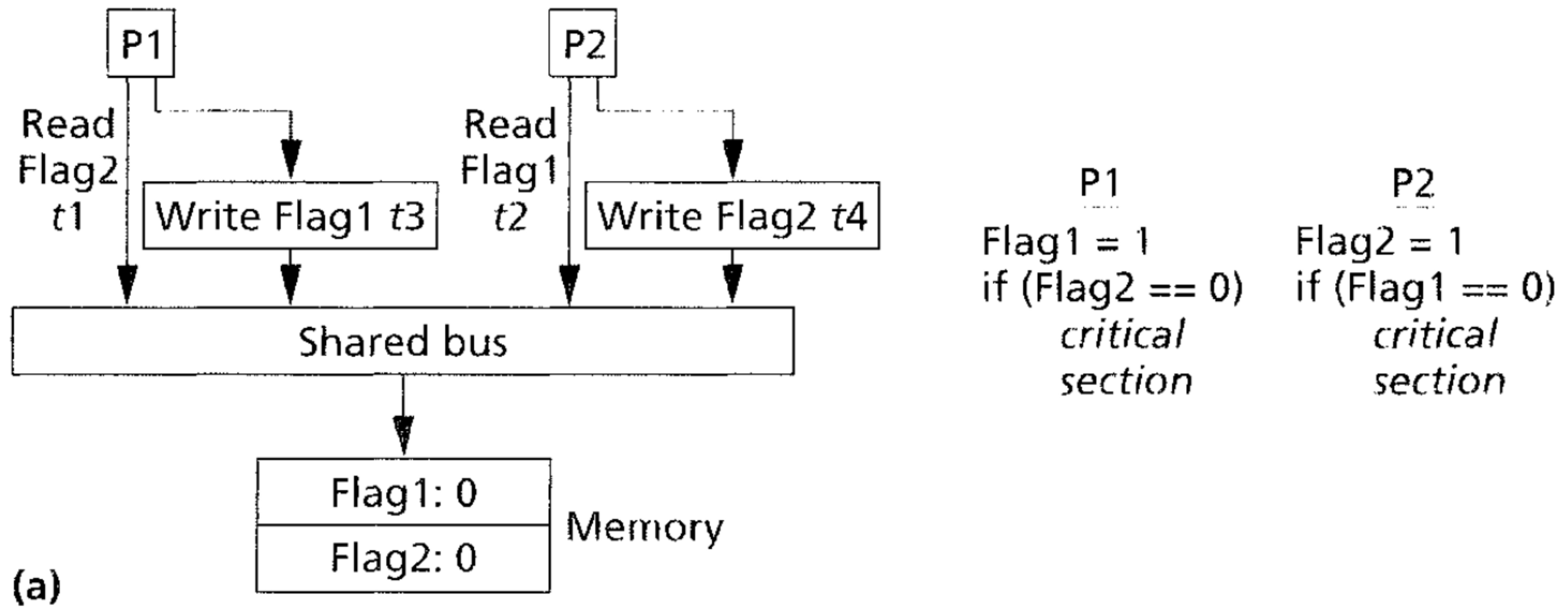
Hardware might make things worse

```
int flag1 = 0, flag2 = 0;

int main () {
    tid id = thread_create (p1, NULL);
    p2 (); thread_join (id);
}

void p1 (void *ignored) {
    flag1 = 1;
    if (!flag2) {
        critical_section_1 ();
    }
}

void p2 (void *ignored) {
    flag2 = 1;
    if (!flag1) {
        critical_section_2 ();
    }
}
```

* this is pseudocode

Can both "critical sections" run?

Maybe, if the hardware works like the following:



https://sadve.cs.illinois.edu/Publications/computer96.pdf

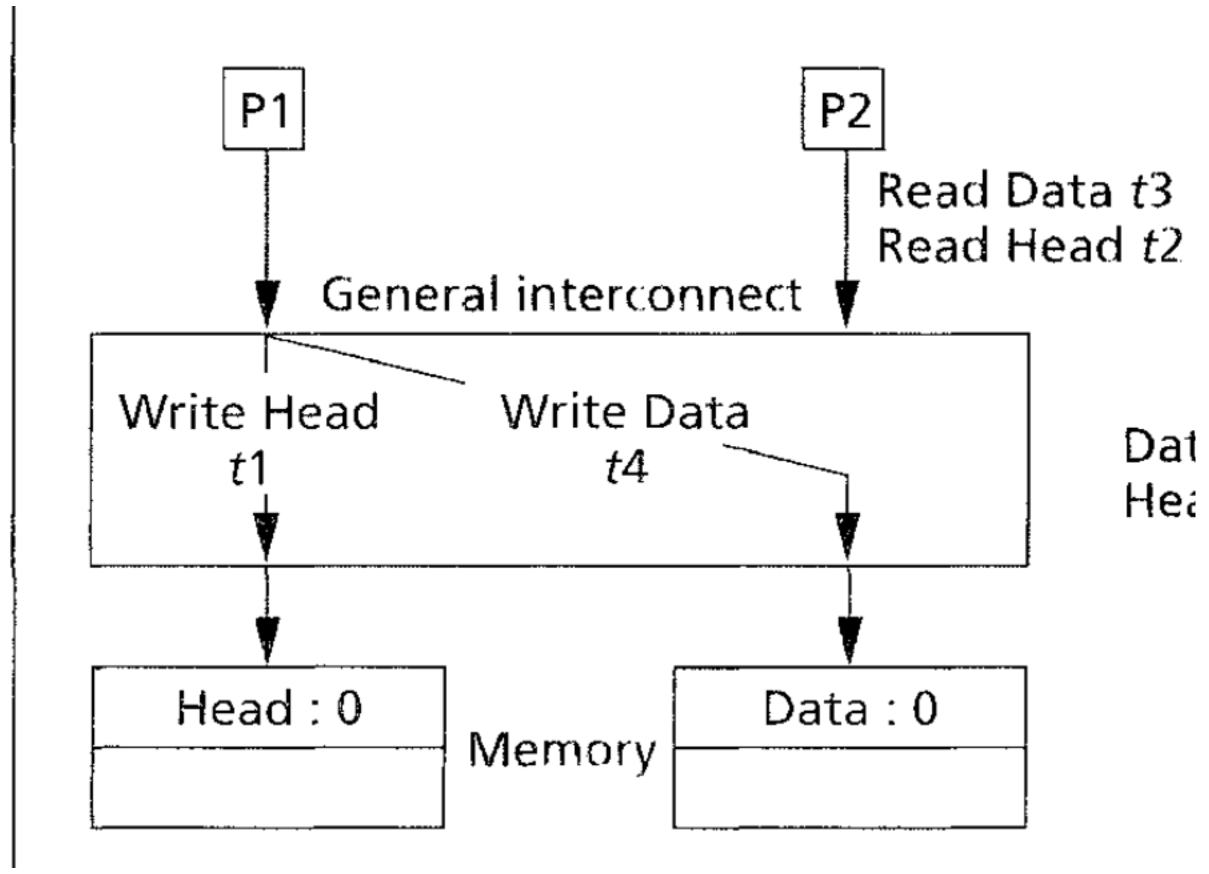
Hardware might make things worse

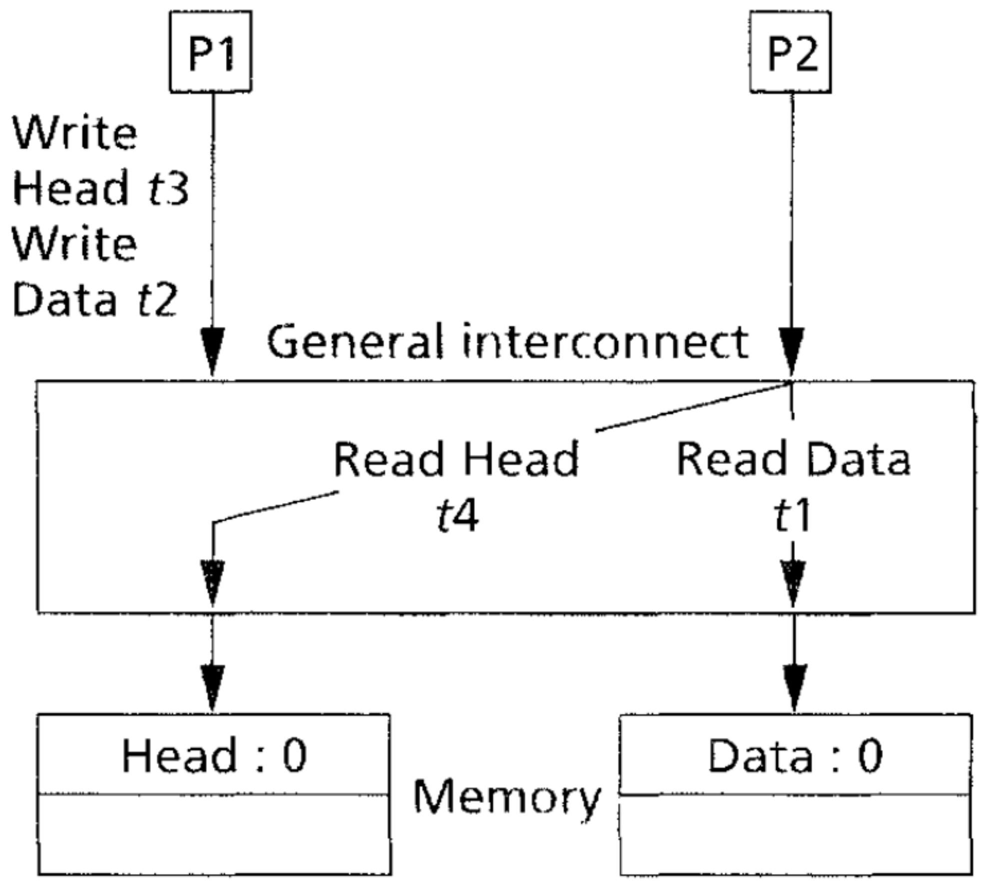
```
int data = 0, head = 0;

void p1 () {
    data = 2000;
    head = 1;
}

int p2 () {
    while (!head) {}
    use(data);
}
```

* this is pseudocode

Can use() be called with value 0, if p2 and p1 run concurrently?

Maybe, if the hardware works like the following:





https://sadve.cs.illinois.edu/Publications/computer96.pdf

Hardware might make things worse

```
int a = 0, b = 0;

void p1 (void *ignored) { a = 1; }

void p2 (void *ignored) {
    if (a == 1) b = 1;
}

void p3 (void *ignored) {
    if (b == 1) use (a);
}
```

* this is pseudocode

Can use() be called with value 0?

Maybe, if the hardware works like the following:

Initially A = B = 0

P1	P2	P3
A = 1		
	if (A == 1)	
	B = 1	
		if (B == 1)
		register1 = A

Result: B = 1, register1 = 0

Certain hardware allows P2 to return the value of P1's write before the write is visible to P3

Reasoning about Concurrency is Hard!

https://sadve.cs.illinois.edu/Publications/computer96.pdf

Don’t worry about hardware-related issues, for now

(Unless explicitly relax it) We assume **sequential consistency** in this class

(On each individual processors)
Writes to each memory location happen in the order that they are issued

Lab 2 is Released Today!

Quiz Time!

Please submit both the quiz booklet and the answer sheet (with your name on both of them!)