



CSE 451: Operating Systems, 26au

L7: Threads

Instructor: Naomi Alterman

Jonathan Trinh

TAs:

Jonathan Trinh (Head)

Mahir Emran

Ted Guan

Jai Rungrachakanont

Soham Raut

Sagar Saby

Andrew Wood

Adapted from slides by Tom Anderson

Administrivia

- Section slides out ahead of time on website, for those interested

Threads

Concurrency: Multiprocessing

- We've been discussing a form of **concurrency** called **multiprocessing**
 - Our OS gets multiple things done concurrently by giving each task its own virtual address space and periodically switching the context with a timer interrupt
- What if we don't need all that memory isolation?
- What if we need to **not** have memory isolation?
 - Electron apps (eg, vscode) work this way: one thread runs a javascript interpreter, the other thread runs native C code to do I/O

Concurrency: Threads

- Allow multiple tasks to share the same address space
 - Each task is called a **thread**
 - Each thread has **its own stack**
 - All threads in a process **share the heap, globals and code segment**
 - Kernel threads run in kernel space, but otherwise the same principle
 - Thread **scheduling is handled by the OS kernel**, just like processes

Déjà vu?

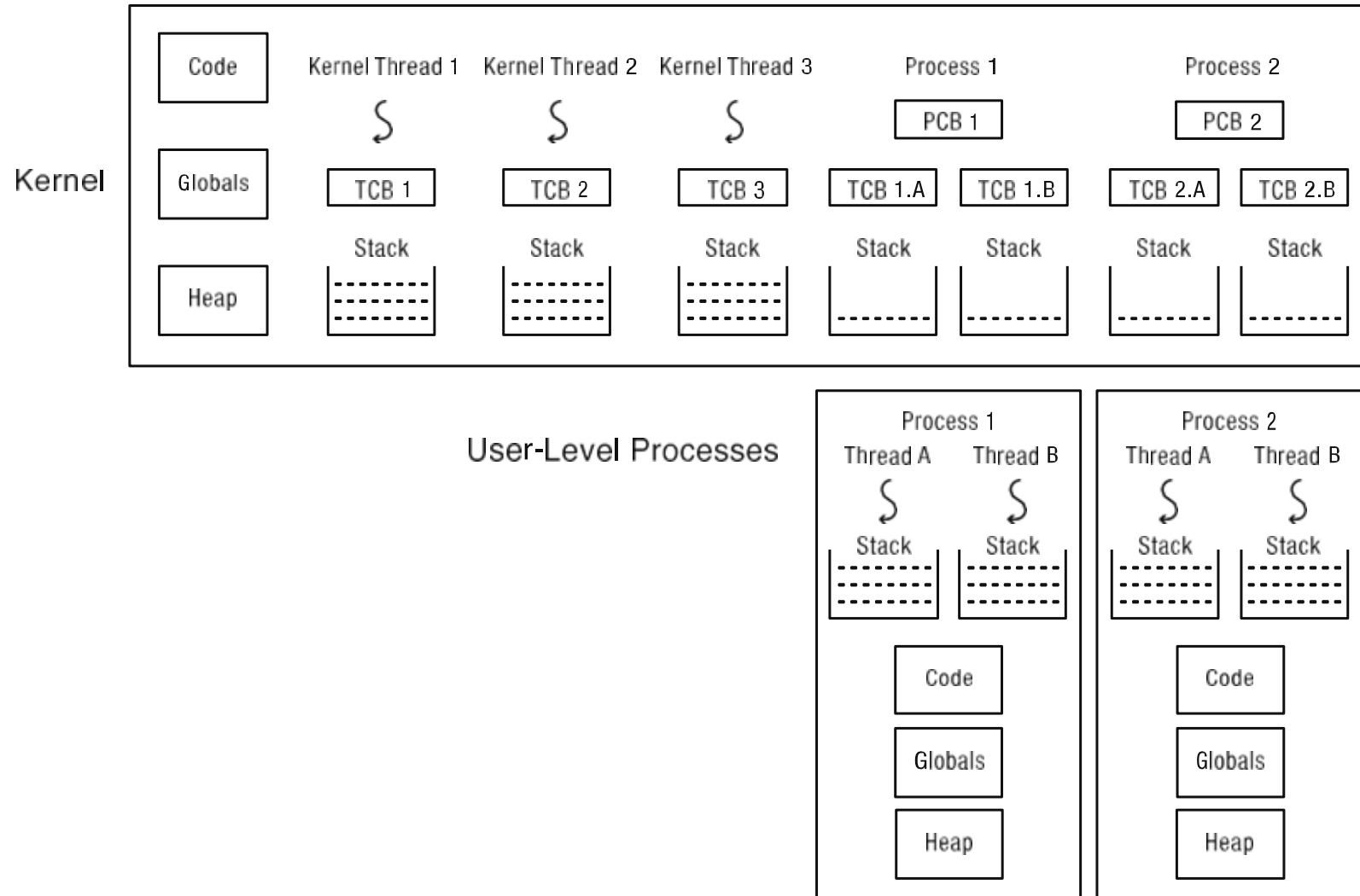
Didn't we learn all about concurrency in
CSE 332 and CSE 333?



Déjà vu?

- Didn't we learn all about concurrency in CSE 332/333?
 - More practice
 - Realistic examples, especially in the project
 - Design patterns and pitfalls
 - Methodology for writing correct concurrent code
 - Implementation
 - How do threads work at the machine level?
 - CPU scheduling
 - If multiple threads to run, which do we do first?

Threading model



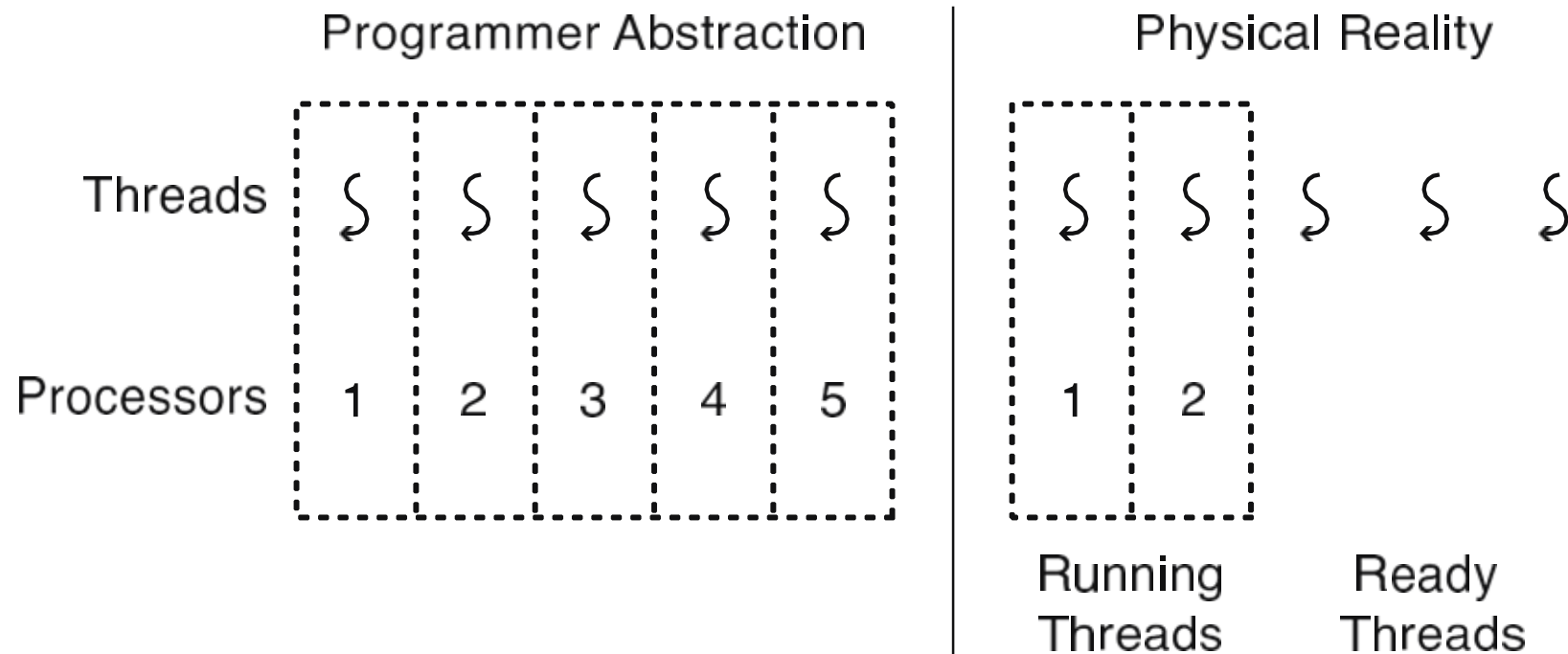


OS as referee

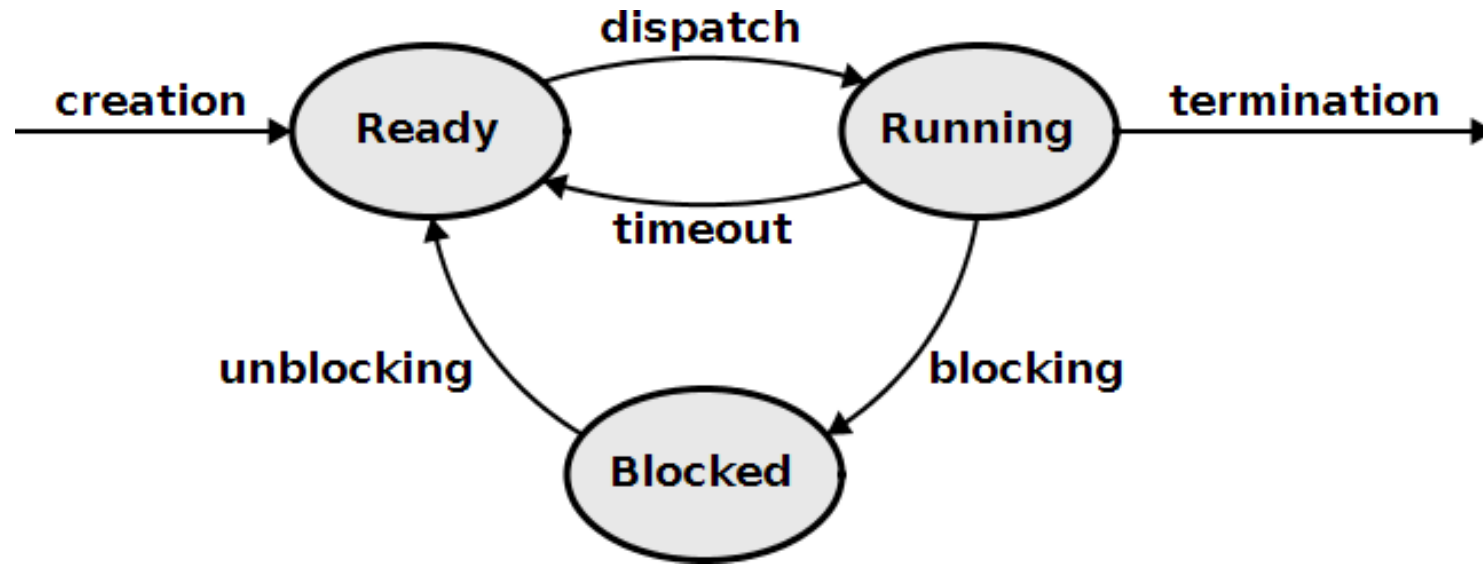
- What if a thread puts too many procedures on its stack?
 - What happens in Java?
 - What happens in the Linux kernel?
 - What happens in xk?
 - What *should* happen?

? OS as illusionist, but only to an extent...

- Infinite number of processors
- Threads execute with variable speed
- Programs must be designed to work with any schedule



~~Process~~ Thread Lifetime



Thread Syscalls

- **thread_create(thread, func, args)**
 - Create a new thread to run func(args)
- **thread_yield()**
 - Relinquish CPU time voluntarily
- **thread_join(thread)**
 - In parent, wait for child thread to exit, then return
- **thread_exit()**
 - Quit thread and clean up, wake up joiner if any

Implementing threads

- **thread_create(func, args)**
 - Allocate thread control block (**TCB**)
 - Allocate user-space stack
 - Build stack frame for base of user stack (stub)
 - Put func, args on stack
 - Put thread on ready list
 - Will run sometime later (maybe right away? maybe not!??)
- **stub(func, args):**
 - Call (*func)(args)
 - If func returns, call thread_exit()

Planting a thread

- Setting up a new thread by hand is a little subtle
- We don't technically start at in the thread's function; we start with an OS-installed wrapper
 - The init process does this too! See “preface” in `user/lib/preface.S`
- We enter the stub with **switch.S**, passing in a hand-constructed trap frame that “resumes” into the new thread

Planting a thread

```
int main () {  
    // ...  
    thread_create(&fun, args);  
    // ...  
}
```

1 →

stub:

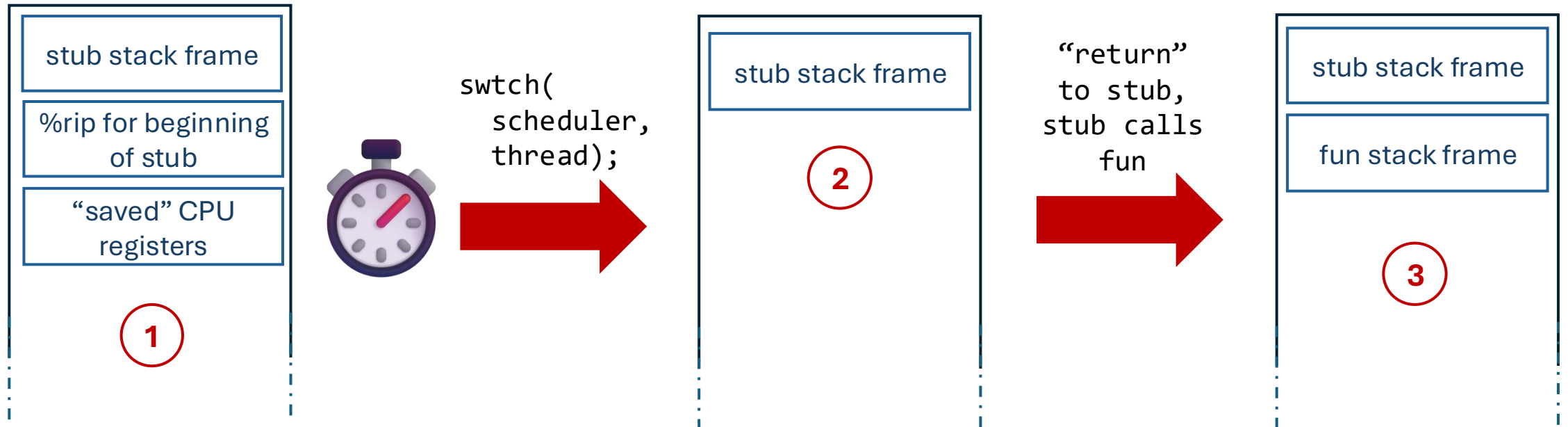
```
    call fun  
    call thread_exit
```

3 →

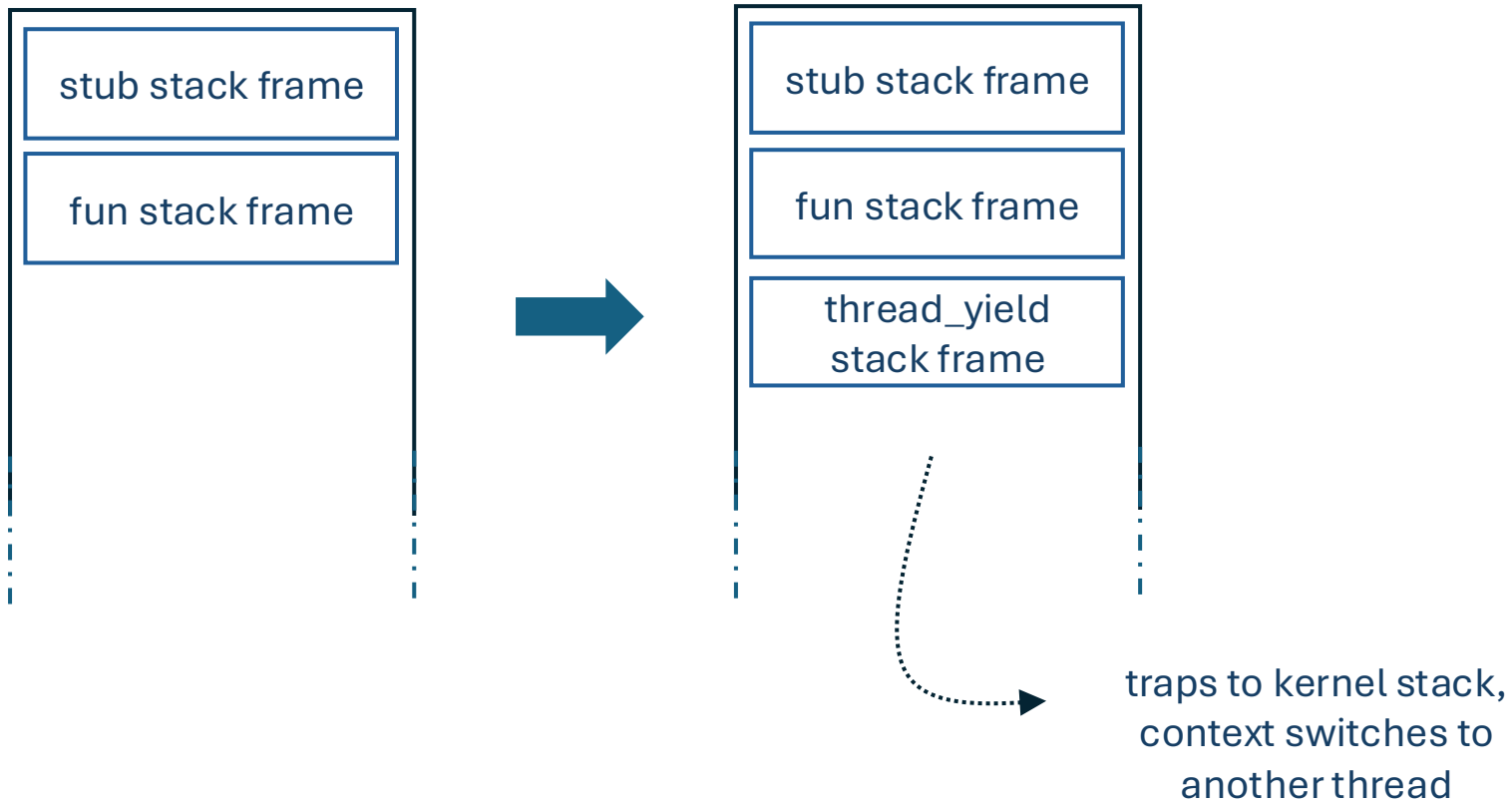
swtch:

```
    # ..save scheduler regs..  
    mov %rsp, (%rdi)  
    mov %rsi, %rsp  
    # ..pop stub regs..
```

2 →



Yielding a thread



Two Threads Call Yield

Thread 1's instructions

"return" from thread_switch
into stub
call go
call thread_yield
choose another thread
call thread_switch
save thread 1 state to TCB
load thread 2 state

return from thread_switch
return from thread_yield
call thread_yield
choose another thread
call thread_switch

Thread 2's instructions

"return" from thread_switch
into stub
call go
call thread_yield
choose another thread
call thread_switch
save thread 2 state to TCB
load thread 1 state

Processor's instructions

"return" from thread_switch
into stub
call go
call thread_yield
choose another thread
call thread_switch
save thread 1 state to TCB
load thread 2 state
"return" from thread_switch
into stub
call go
call thread_yield
choose another thread
call thread_switch
save thread 2 state to TCB
load thread 1 state
return from thread_switch
return from thread_yield
call thread_yield
choose another thread
call thread_switch

Synchronization

Haha uh oh

- **NB:** concurrency is hard
 - Humans are not very good at keeping track of multiple things happening simultaneously
 - *Any* concurrency (processes, threads) means multiple syscalls can be running at once
 - Multiple syscalls running at once means we need to protect shared kernel data from getting corrupted

Question: Can this panic?

// Thread 1

```
float theta = 0;  
theta = promptFloat();  
ready = true;
```

// Thread 2

```
while (!ready) { }  
val = cos(theta);  
if (val != cos(theta)) {  
    panic()  
}
```

Motivation

- C++ (and most other systems languages) state that memory addresses being concurrently read or written produce ***undefined behavior*** ([link](#))
 - Literally, we can't know what code the compiler will generate
 - Hardware behavior is architecture-specific and in most cases not practical to predict
 - Software optimizations are very limited if we assume the “worst case” that any variable can change at any time (so the compiler *doesn't* assume that)
- **Synchronization primitives** (from the OS's point-of-view):
A **small set of operations** to work safely on **concurrently shared memory**
 - We are *super-careful* to get it right
 - Implementation is likely hardware-specific

Example: threadHello

```
#define NTHREADS 10
thread_t threads[NTHREADS];
Void main() {
    for (i = 0; i < NTHREADS; i++) {
        thread_create(&threads[i], &go, i);
    }
    for (i = 0; i < NTHREADS; i++) {
        exitValue = thread_join(threads[i]);
        printf("Thread %d returned with %ld\n", i, exitValue);
    }
    printf("Main thread done.\n");
}
void go (int n) {
    printf("Hello from thread %d\n", n);
    thread_exit(100 + n);
}
```

```
bash-3.2$ ./threadHello
Hello from thread 0
Hello from thread 1
Thread 0 returned 100
Hello from thread 3
Hello from thread 4
Thread 1 returned 101
Hello from thread 5
Hello from thread 2
Hello from thread 6
Hello from thread 8
Hello from thread 7
Hello from thread 9
Thread 2 returned 102
Thread 3 returned 103
Thread 4 returned 104
Thread 5 returned 105
Thread 6 returned 106
Thread 7 returned 107
Thread 8 returned 108
Thread 9 returned 109
Main thread done.
```

Programmer vs. Processor View

Programmer's View

.
.
.
x = x + 1;
y = y + x;
z = x + 5y;
.
.
.

Possible Execution #1

.
.
.
x = x + 1;
y = y + x;
z = x + 5y;
.
.
.

Possible Execution #2

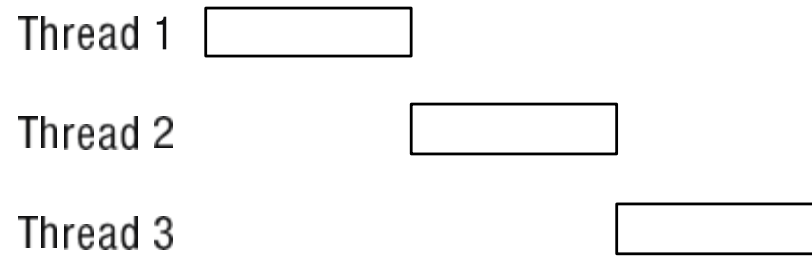
.
.
.
x = x + 1;
.....
Thread is suspended.
Other thread(s) run.
Thread is resumed.
.....
y = y + x;
z = x + 5y;

Possible Execution #3

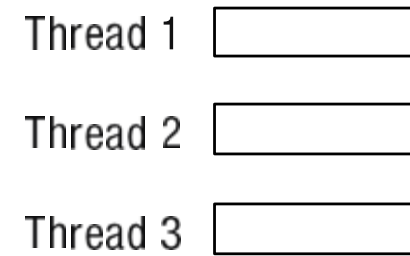
.
.
.
x = x + 1;
y = y + x;
.....
Thread is suspended.
Other thread(s) run.
Thread is resumed.
.....
z = x + 5y;

Possible Executions

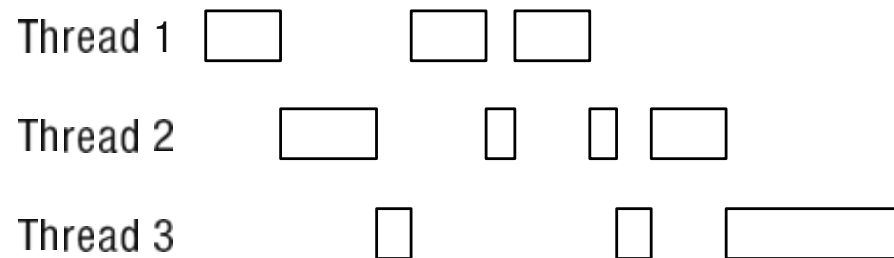
One Execution



Another Execution



Another Execution



Definitions

Race condition: output of a concurrent program depends on the order of operations between threads

Data race: when a thread can read or modify a memory location while another thread may be modifying it. (2 threads, 1 is a write, no intervening synchronization)

Memory Fence: hardware instruction to enforce ordering

- All instructions before fence complete before any started after fence

More Definitions

Mutual exclusion: only one thread does a particular thing at a time

Critical section: piece of code that only one thread can execute at once

Lock: prevent someone from doing something

- Lock before entering critical section, before accessing shared data
- Unlock when leaving, after done accessing shared data
- Wait if locked (all synchronization involves waiting!)

Concurrency landscape

	Concurrent Applications		
Software primitives to allow concurrency (built with the rows below)	Semaphores	Locks	Condition Variables
Hardware primitives to prevent concurrency	Interrupt Disabling	Atomic memory instructions	
Sources of concurrency	Multiple CPU cores (or processors)	Hardware Interrupts	Kernels with context switching

Concurrency goals

- **Safety:** make sure bad things don't happen
- **Liveness:** make sure that, despite safety, good things still do happen

Too Much Milk Example

	Person A	Person B
12:30	Look in fridge. Out of milk.	
12:35	Leave for store.	
12:40	Arrive at store.	Look in fridge. Out of milk.
12:45	Buy milk.	Leave for store.
12:50	Arrive home, put milk away.	Arrive at store.
12:55		Buy milk.
1:00		Arrive home, put milk away. <i>Oh no!</i>

Too Much Milk, #1

- Correctness property
 - Someone buys if needed (liveness)
 - At most one person buys (safety)

- Try #1: leave a note

→ if (!note)

→ if (!milk) {
 leave note
 buy milk
 remove note
}

Too Much Milk, #2

Thread A



```
leave note A
if (!note B) {
    if (!milk) buy milk
}
remove note A
```

Thread B



```
leave note B
if (!noteA) {
    if (!milk) buy milk
}
remove note B
```

Too Much Milk, #3

Thread A

```
leave note A
while (note B) { } // X
if (!milk) {
    buy milk;
}
remove note A
```

Can guarantee at X and Y that either:

- (i) Safe for me to buy
- (ii) Other will buy, ok to quit

Thread B

```
leave note B
if (!noteA) { // Y
    if (!milk) {
        buy milk
    }
}
remove note B
```

Lessons

- Solution is complicated
 - “obvious” code often has bugs
- Modern compilers/architectures reorder instructions
 - Making reasoning even more difficult
- Generalizing to many threads/processors
 - Even more complex: see Peterson’s algorithm

Class concluded here on 10/14
and will resume on 10/16



Locks

A solution to concurrency tragedy

Locks

Lock::acquire: wait until lock is free, then take it

Lock::release: release lock, waking up anyone waiting for it

1. At most one lock holder at a time (safety)
2. If no one holding, acquire gets lock (liveness)
3. If all lock holders finish and no higher priority waiters, waiter eventually gets lock (liveness)

Too Much Milk, #4

Locks allow concurrent code to be much simpler:

```
milk_lock.acquire();  
if (!milk) {  
    buy milk  
}  
milk_lock.release();
```

Lock Example: Malloc/Free

```
char *malloc (n) {  
    heaplock.acquire();  
    p = allocate memory  
    heaplock.release();  
    return p;  
}
```

```
void free(char *p) {  
    heaplock.acquire();  
    put p back on free list  
    heaplock.release();  
}
```

Rules for Using Locks

- Lock is initially free
- Always acquire before accessing shared data structure
 - Beginning of procedure!
- Always release after finishing with shared data
 - End of procedure!
 - Only the lock holder can release
 - DO NOT throw lock for someone else to release
- Never access shared data without lock
 - Danger!

Danger!! Bad! Double Checked Locking

```
if (p == NULL) {  
    lock.acquire();  
    if (p == NULL) {  
        p = newP();  
    }  
    lock.release();  
}  
use p->field1
```

```
newP() {  
    tmp = malloc(sizeof(p));  
    tmp->field1 = ...  
    tmp->field2 = ...  
    return tmp;  
}
```

Single Checked Locking

```
lock.acquire();  
if (p == NULL) {  
    p = newP();  
}  
lock.release();  
use p->field1
```

```
newP() {  
    tmp = malloc(sizeof(p));  
    tmp->field1 = ...  
    tmp->field2 = ...  
    return tmp;  
}
```

Question: Why only Acquire/Release?

Suppose we add a method to a lock, to ask if the lock is free.

Suppose it returns true. Is the lock:

- Free?
- Busy?
- Don't know?