CS 423
Operating System Design: Concurrency

Tianyin Xu

* Thanks for Prof. Adam Bates for the slides.

Concurrency vs Parallelism

Two tasks

1. Get a visa

2. Prepare slides

1. Sequential execution
2. Concurrent execution
3. Parallel execution
4. Concurrent but not parallel
5. Parallel but not concurrent
6. Parallel and concurrent

Why Concurrency?

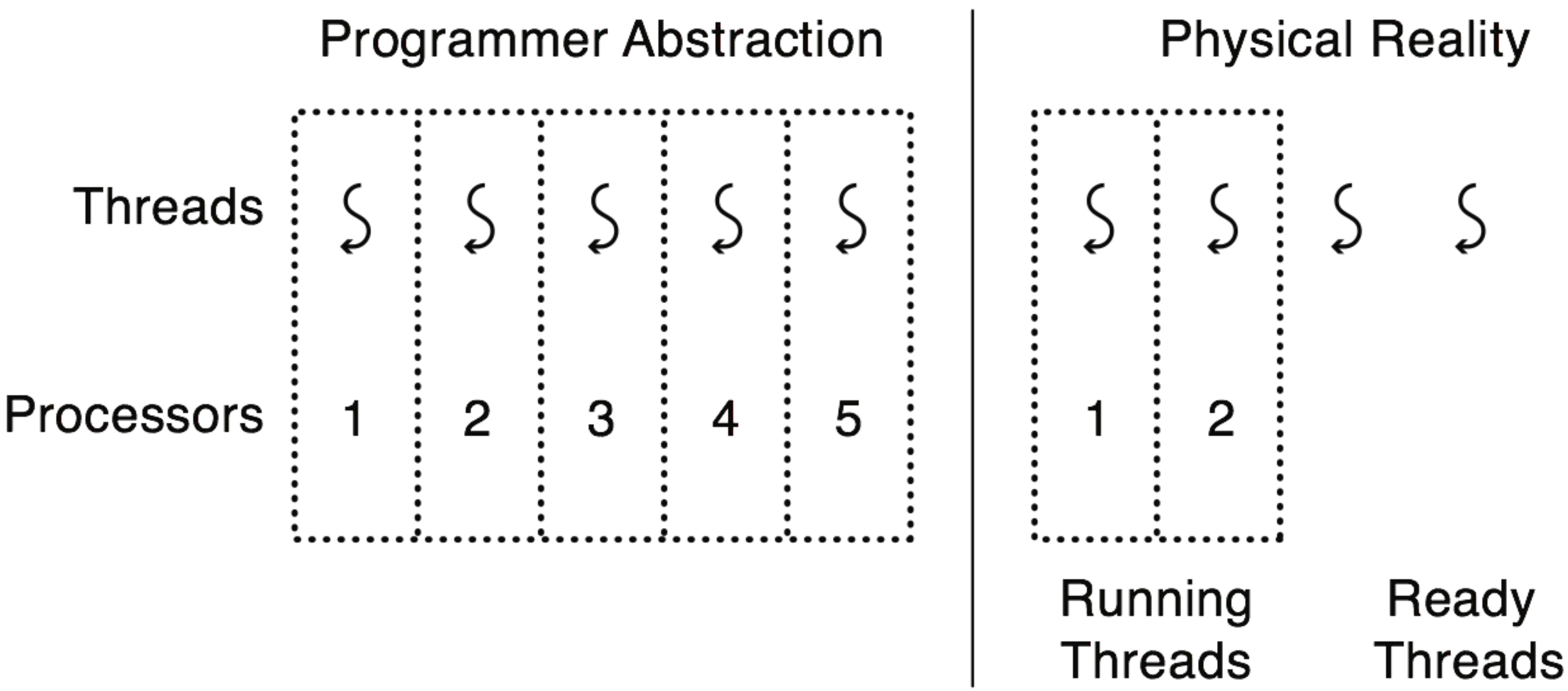
- Servers
  - Multiple connections handled simultaneously
- Parallel programs
  - To achieve better performance
- Programs with user interfaces
  - To achieve user responsiveness while doing computation
- Network and disk bound programs
  - To hide network/disk latency

Definitions

- Thread: A single execution sequence that represents a separately schedulable task.
  - *Single execution sequence*: intuitive and familiar programming model
  - *separately schedulable*: OS can run or suspend a thread at any time.
  - Schedulers operate over threads/tasks, both kernel and user threads.
- *Does the OS protect all threads from one another?*

The Thread Abstraction

- Infinite number of processors
- Threads execute with variable speed



Programmer vs. Processor View

Programmer View

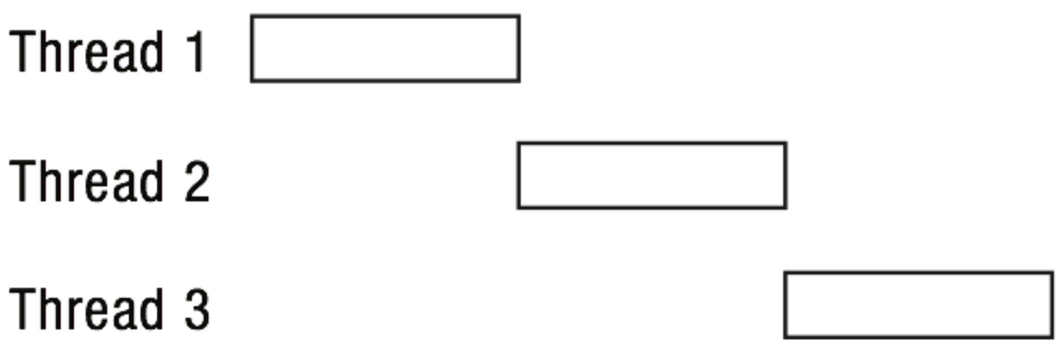
Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
.	.	.	.
x = x + 1;	x = x + 1;	x = x + 1;	x = x + 1;
y = y + x;	y = y + x;	....	y = y + x;
z = x + 5y;	z = x + 5y;	Thread is suspended.	....
.	.	Other thread(s) run.	Thread is suspended.
.	.	Thread is resumed.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

Variable Speed: Program must anticipate all of these possible executions

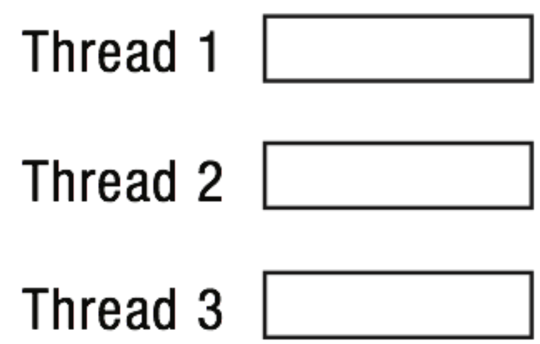
Possible Executions

Processor View

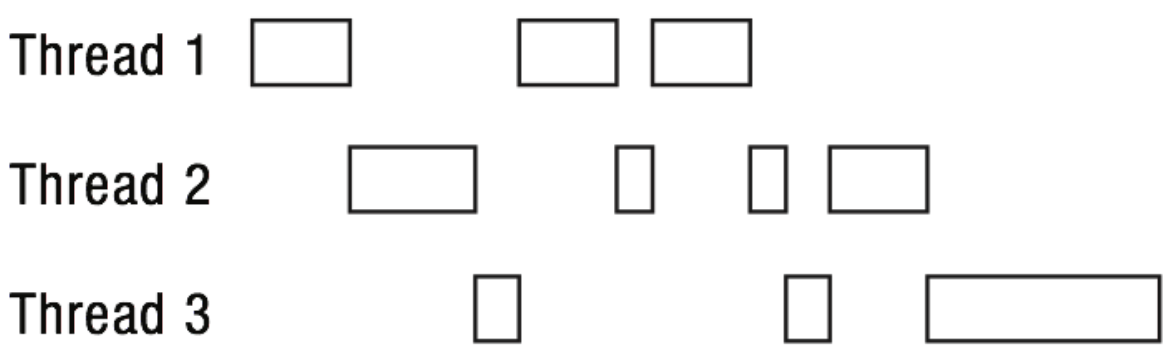
One Execution



Another Execution



Another Execution



Something to look forward to when we discuss scheduling!

Thread Ops

- thread_create(thread, func, args)
  Create a new thread to run func(args)

- thread_yield()
  Relinquish processor voluntarily

- thread_join(thread)
  In parent, wait for forked thread to exit, then return

- thread_exit
  Quit thread and clean up, wake up joiner if any

Ex: threadHello

```
#define NTHREADS 10
thread_t threads[NTHREADS];
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);
    // REACHED?
}
```

Ex: threadHello output

```
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.
```

- Must “thread returned” print in order?
- What is maximum # of threads that exist when thread 5 prints hello?
- Minimum?
- Why aren’t any messages interrupted mid-string?

Create/Join Concurrency

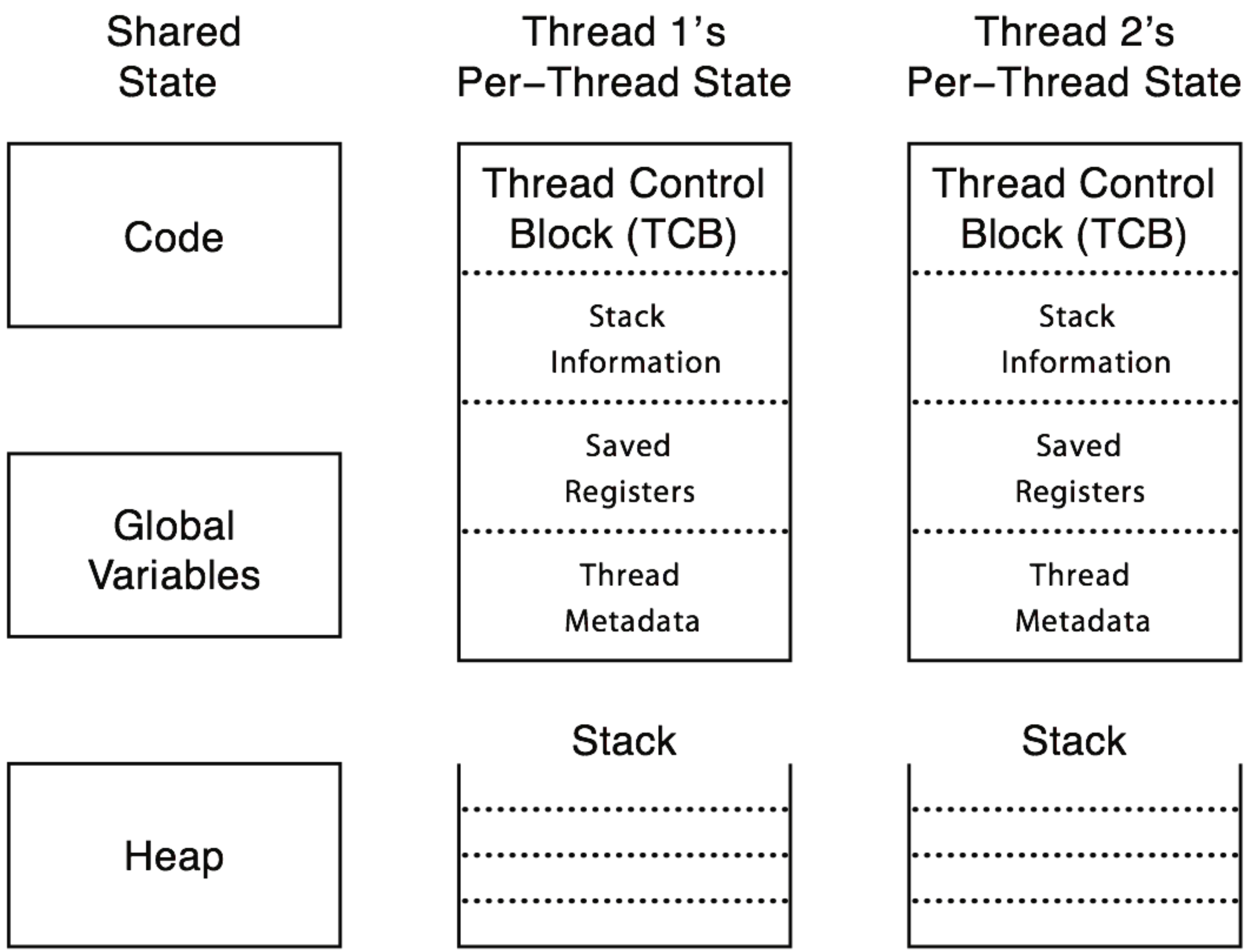
- Threads can create children, and wait for their completion
- Data only shared before fork/after join
- Examples:
  - Web server: fork a new thread for every new connection
    - As long as the threads are completely independent
  - Merge sort
  - Parallel memory copy

Ex: bzero

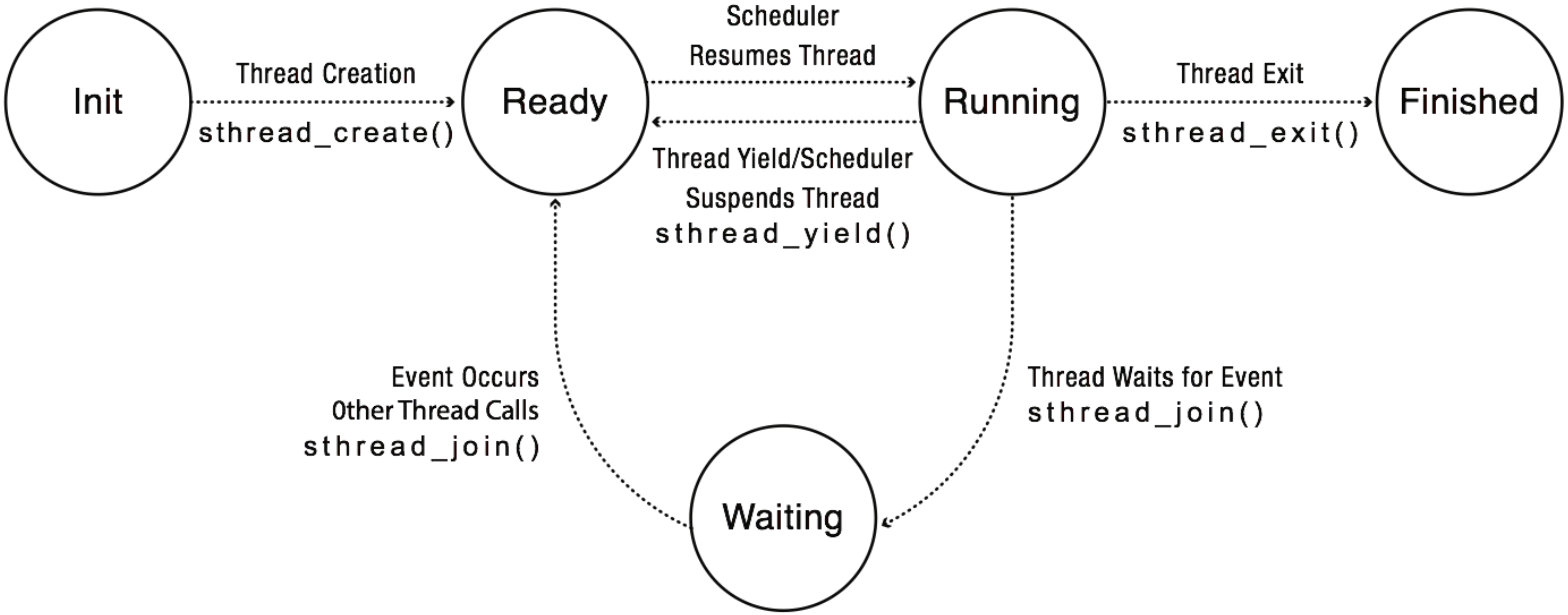
```
void blockzero (unsigned char *p, int length) {
    int i, j;
    thread_t threads[NTHREADS];
    struct bzeroparams params[NTHREADS];

// For simplicity, assumes length is divisible by NTHREADS.
for (i = 0, j = 0; i < NTHREADS; i++, j += length/NTHREADS) {
        params[i].buffer = p + i * length/NTHREADS;
        params[i].length = length/NTHREADS;
        thread_create_p(&(threads[i]), &go, &params[i]);
    }
    for (i = 0; i < NTHREADS; i++) {
        thread_join(threads[i]);
    }
}
```

Thread Data Structures



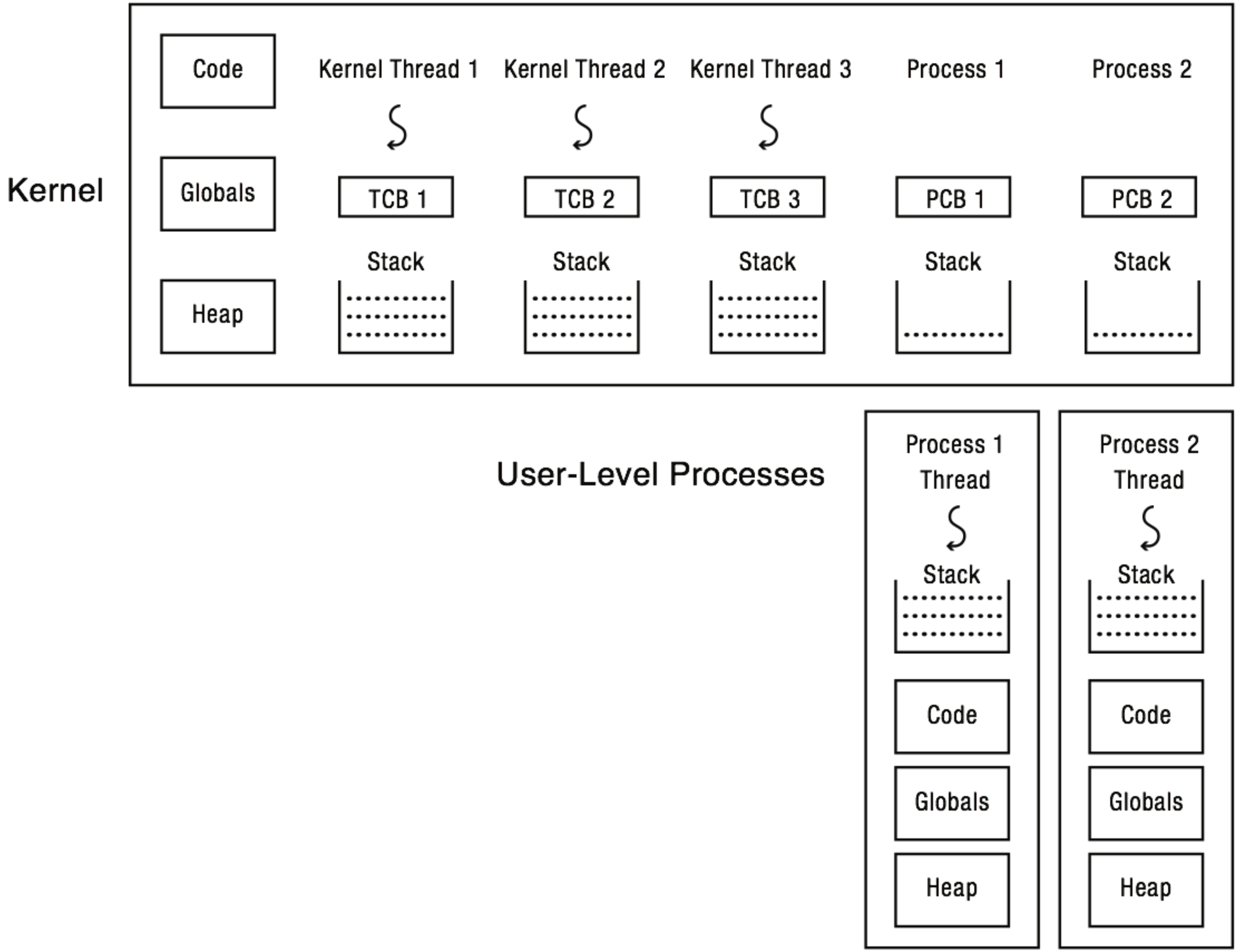
Thread Lifecycle



Thread Implementations

- Kernel threads
  - Thread abstraction only available to kernel
  - To the kernel, a kernel thread and a single threaded user process look quite similar
- Multithreaded processes using kernel threads
  - Kernel thread operations available via syscall
- User-level threads
  - Thread operations without system calls

Multithreaded OS Kernel



Implementing Threads

- Thread_fork(func, args)
  - Allocate thread control block
  - Allocate stack
  - Build stack frame for base of stack (stub)
  - Put func, args on stack
  - Put thread on ready list
  - Will run sometime later (maybe right away!)

- stub(func, args):
  - Call (*func)(args)
  - If return, call thread_exit()

Implementing Threads

- Thread_Exit
  - Remove thread from the ready list so that it will never run again
  - Free the per-thread state allocated for the thread

Ex: Two Threads call Yield

Thread 1's instructions	Thread 2's instructions	**Processor's instructions**
"return" from thread_switch into stub		"return" from thread_switch into stub
call go		call go
call thread_yield		call thread_yield
choose another thread		choose another thread
call thread_switch		call thread_switch
save thread 1 state to TCB		save thread 1 state to TCB
load thread 2 state		load thread 2 state
	"return" from thread_switch into stub	"return" from thread_switch into stub
	call go	call go
	call thread_yield	call thread_yield
	choose another thread	choose another thread
	call thread_switch	call thread_switch
	save thread 2 state to TCB	save thread 2 state to TCB
	load thread 1 state	load thread 1 state
return from thread_switch		return from thread_switch
return from thread_yield		return from thread_yield
call thread_yield		call thread_yield
choose another thread		choose another thread
call thread_switch		call thread_switch

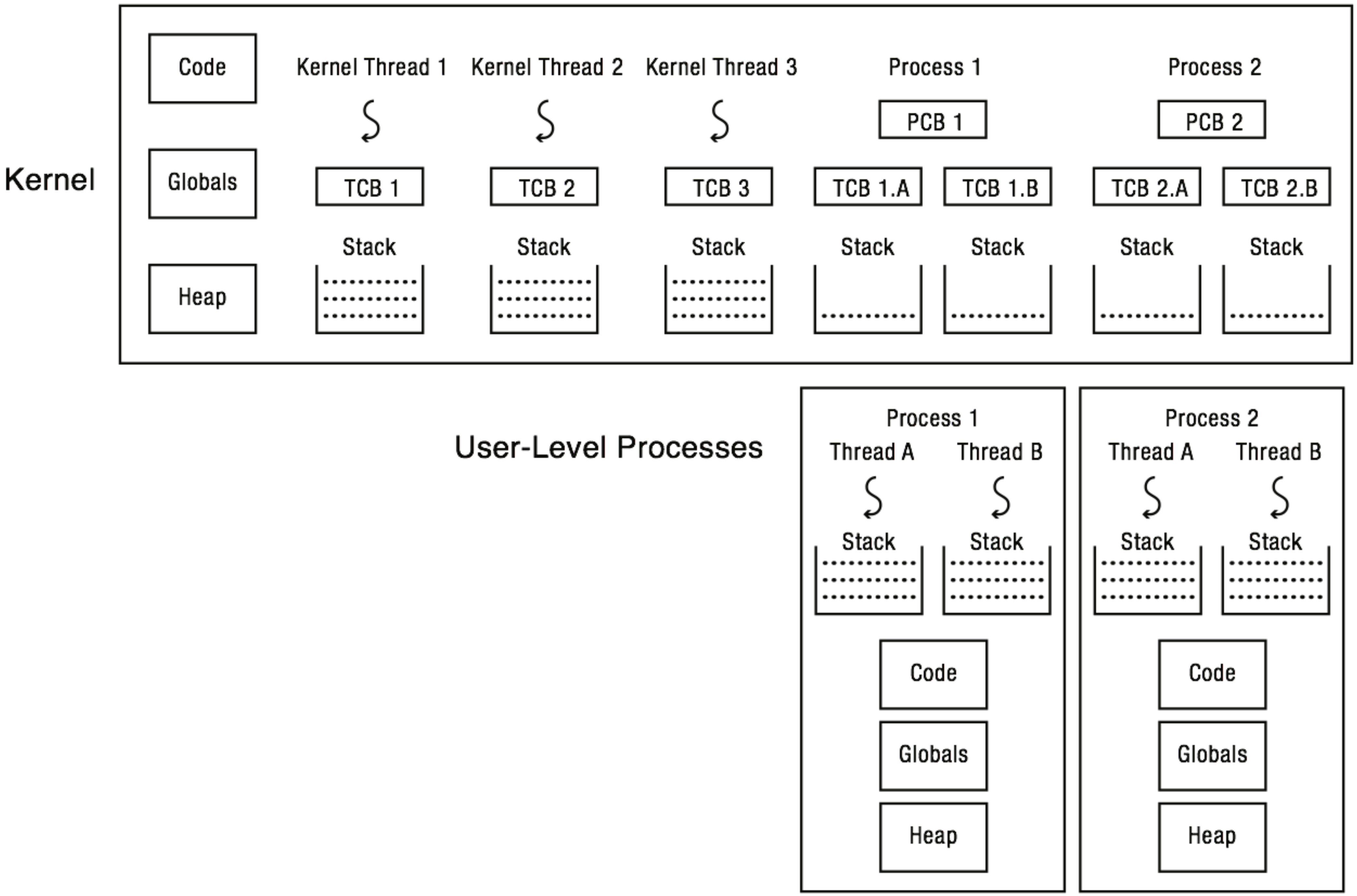
Multi-threaded User Processes

Take 1:

- User thread = kernel thread (Linux, MacOS)
  - System calls for thread fork, join, exit (and lock, unlock,…)
  - Kernel does context switch
  - Simple, but a lot of transitions between user and kernel mode

Multi-threaded User Processes

Take 1:



Multi-threaded User Processes

Take 2:

- Green threads (early Java)
  - User-level library, within a single-threaded process
  - Library does thread context switch
  - Preemption via upcall/UNIX signal on timer interrupt
  - Use multiple processes for parallelism
  - Shared memory region mapped into each process

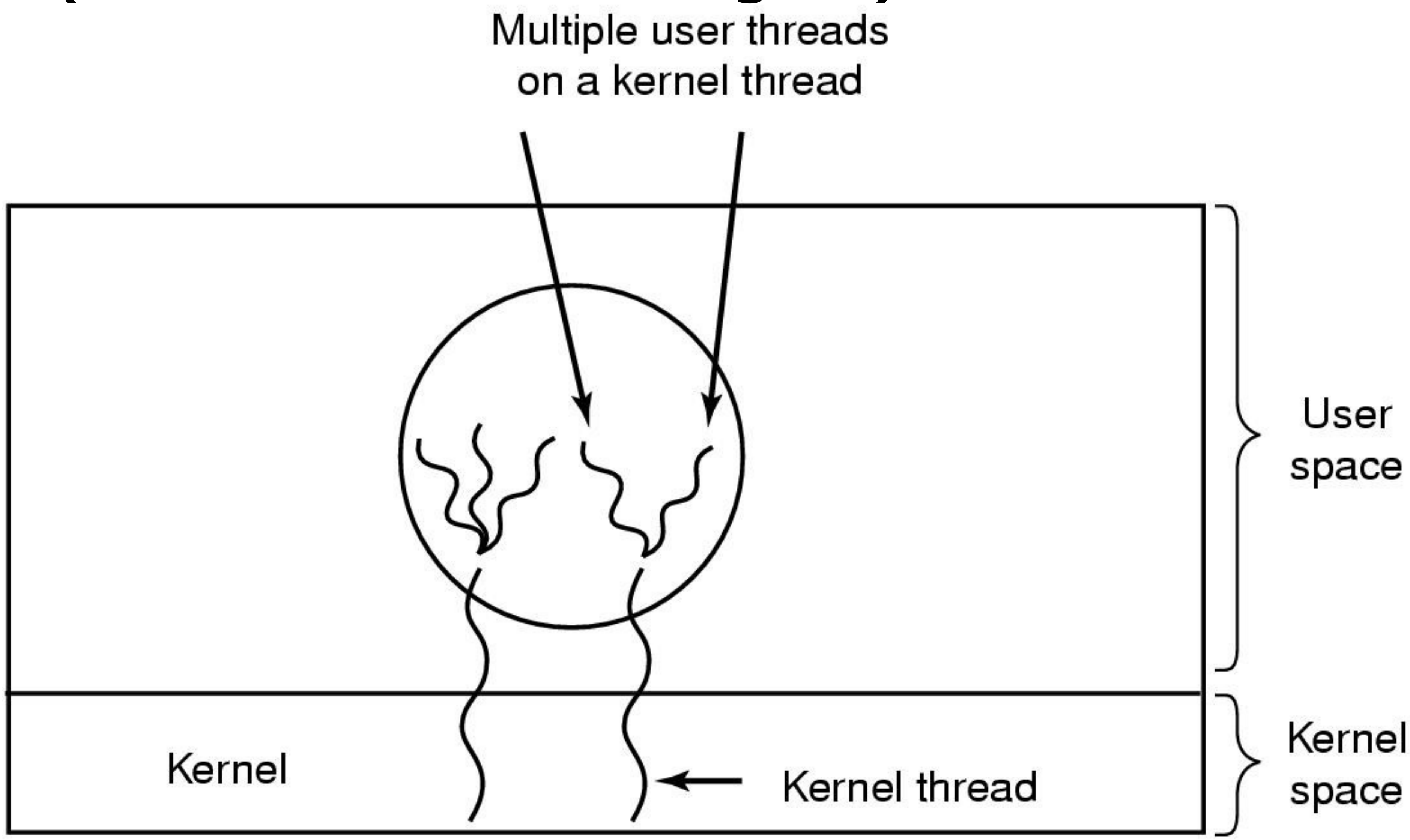
Multi-threaded User Processes

Take 3:

- Scheduler activations (Windows 8):
  - Kernel allocates processors to user-level library
  - Thread library implements context switch
  - Thread library decides what thread to run next
- Upcall whenever kernel needs a user-level scheduling decision:
  - Process assigned a new processor
  - Processor removed from process
  - System call blocks in kernel

Multi-threaded User Processes

Take 3: (What's old is new again)



M:N model multiplexes N user-level threads onto M kernel-level threads

Good idea? Bad Idea?

Question

Compare event-driven programming with multithreaded concurrency. Which is better in which circumstances, and why?