Concurrency

Motivation

- Operating systems need to be able to handle multiple things at once
  - processes, interrupts, background system maintenance
- Servers need to handle MTAO
  - Multiple connections handled simultaneously
- Parallel programs need to handle MTAO
  - To achieve better performance
- Programs with user interfaces often need to handle MTAO
  - To achieve user responsiveness while doing computation
- Network and disk bound programs need to handle MTAO
  - To hide network/disk latency

But we already covered concurrency…

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

Definitions

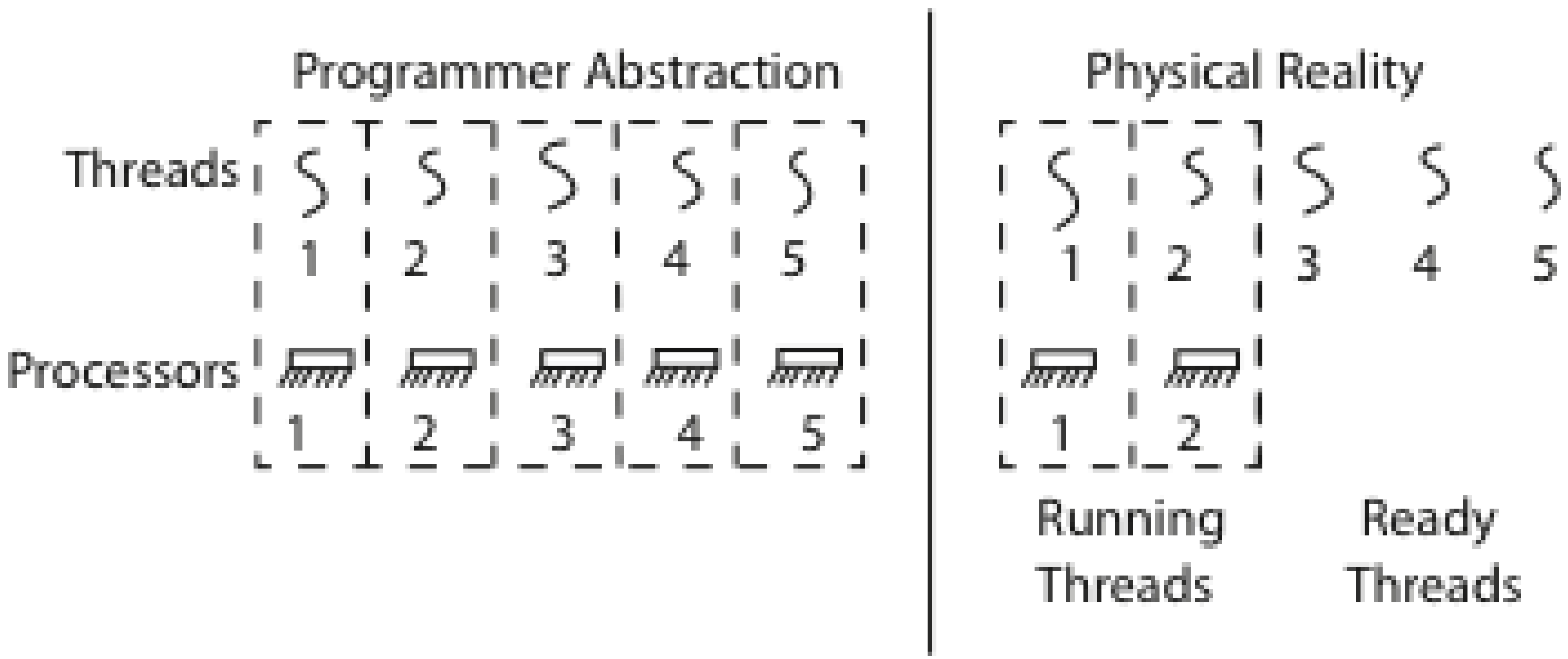
- A thread is a single execution sequence that represents a separately schedulable task
  - Single execution sequence: familiar programming model
  - Separately schedulable: OS can run or suspend a thread at any time
- Protection is an orthogonal concept
  - Can have one or many threads per protection domain

Threads in the Kernel and at User-level

- Multi-threaded kernel
  - multiple threads, sharing kernel data structures, capable of using privileged instructions
  - OS/161 assignment 1
- Multiprocess kernel
  - Multiple single-threaded processes
  - System calls access shared kernel data structures
  - OS/161 assignment 2
- Multi-threaded user program
  - multiple threads, sharing same data structures, isolated from other user processes
- Multiple multi-threaded processes

Thread Abstraction

- Infinite number of processors



Because there aren’t infinite real cores…

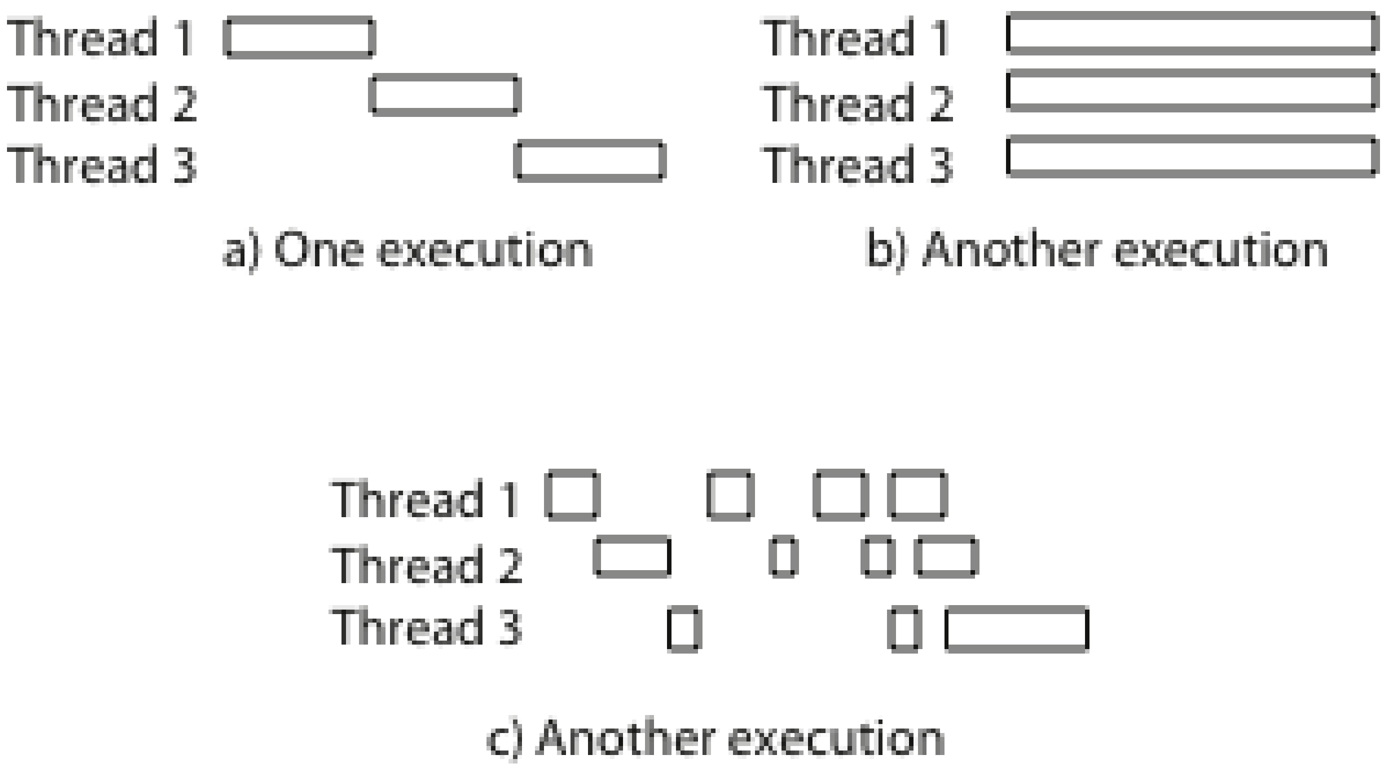
- Each of the infinite abstract processors runs at variable speed

- Programs must be designed to work with any schedule
  - Program correctness doesn’t depend on timing
    - “race free”

Programmer vs. Processor 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

Possible Executions



Thread Operations

- thread_create(thread, func, args)
  - Create a new thread to run func(args)
  - OS/161: thread_fork
- thread_yield()
  - Relinquish processor voluntarily
  - OS/161: thread_yield
- thread_join(thread)
  - In parent, wait for forked thread to exit, then return
  - OS/161: assignment 1
- thread_exit
  - Quit thread and clean up, wake up joiner if any
  - OS/161: thread_exit

Example: threadHello

```
#include <pthread.h>
#include <stdio.h>
#define NTHREADS 10
pthread_t threads[NTHREADS];
void* go(void *arg) {
        long int n = (long int)arg;
        printf("Hello from thread %ld\n", n);
        return (void*)100+n;
}
int main(int argc, char *argv[]) {
        long int I;
        for (i=0; i<NTHREADS; i++) pthread_create(&threads[i], NULL, &go, (void*)i);
        for (i=0; i<NTHREADS; i++) {
                void * exitValue;
                pthread_join(threads[i], &exitValue);
                printf("Thread %ld returned with value %ld\n, I, (long int)exitValue );
        }
        printf("Main thread done");
        return 0;
}
```

```
$ gcc threadHello.c -lpthread
```

Example Output

```
$ ./a.out
Hello from thread 2
Hello from thread 3
Hello from thread 1
Hello from thread 4
Hello from thread 0
Hello from thread 5
Hello from thread 6
Hello from thread 7
Hello from thread 8
Thread 0 returned with value 100
Thread 1 returned with value 101
Thread 2 returned with value 102
Thread 3 returned with value 103
Thread 4 returned with value 104
Thread 5 returned with value 105
Thread 6 returned with value 106
Thread 7 returned with value 107
Thread 8 returned with value 108
Hello from thread 9
Thread 9 returned with value 109
Main thread done
```

- Why aren't the hello msgs in order?
- Why are the"thread returned" msgs in order?
- What is the maximum number of threads running when thread 5 prints hello?
- What is the minimum number?

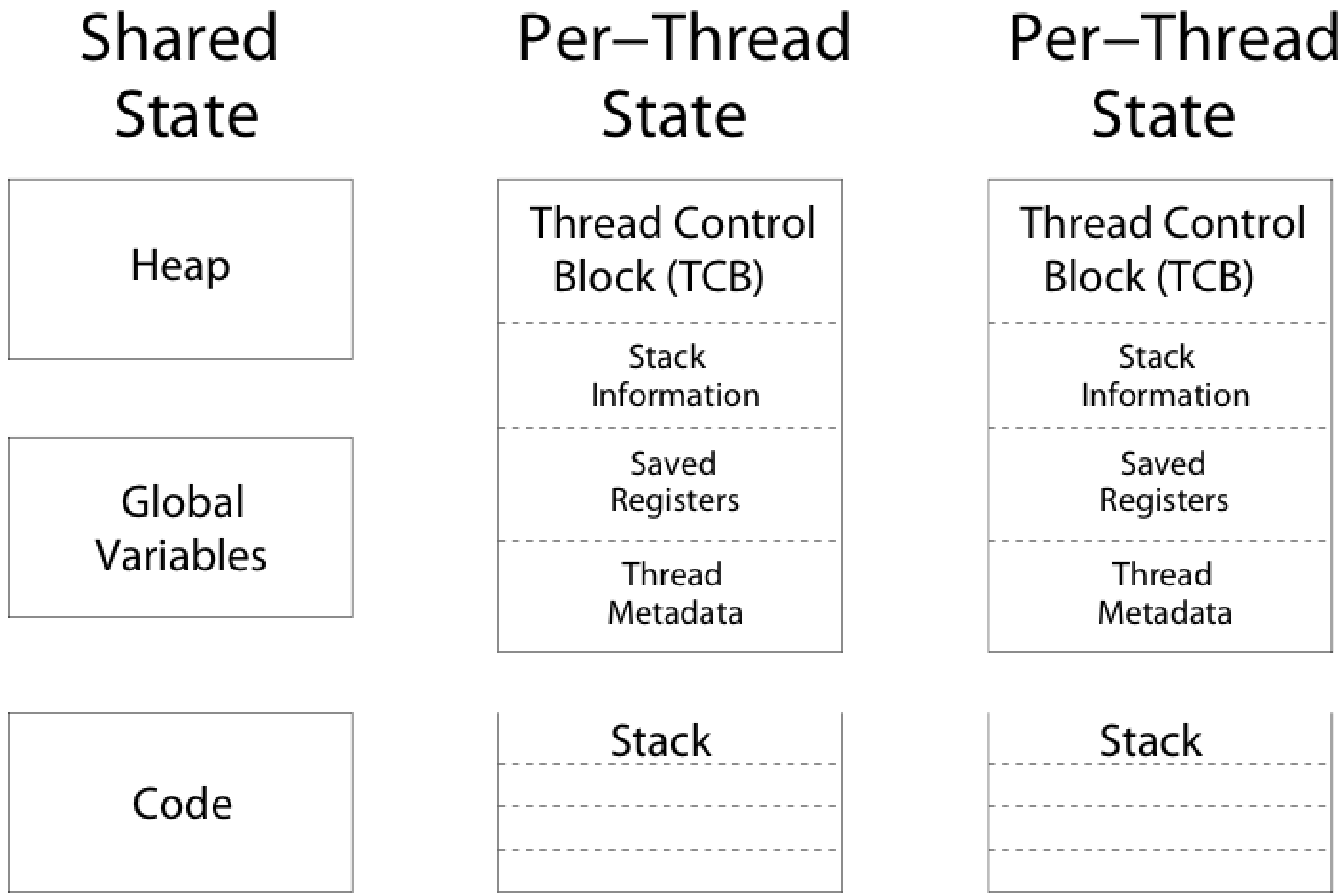
Fork/Join Concurrency

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

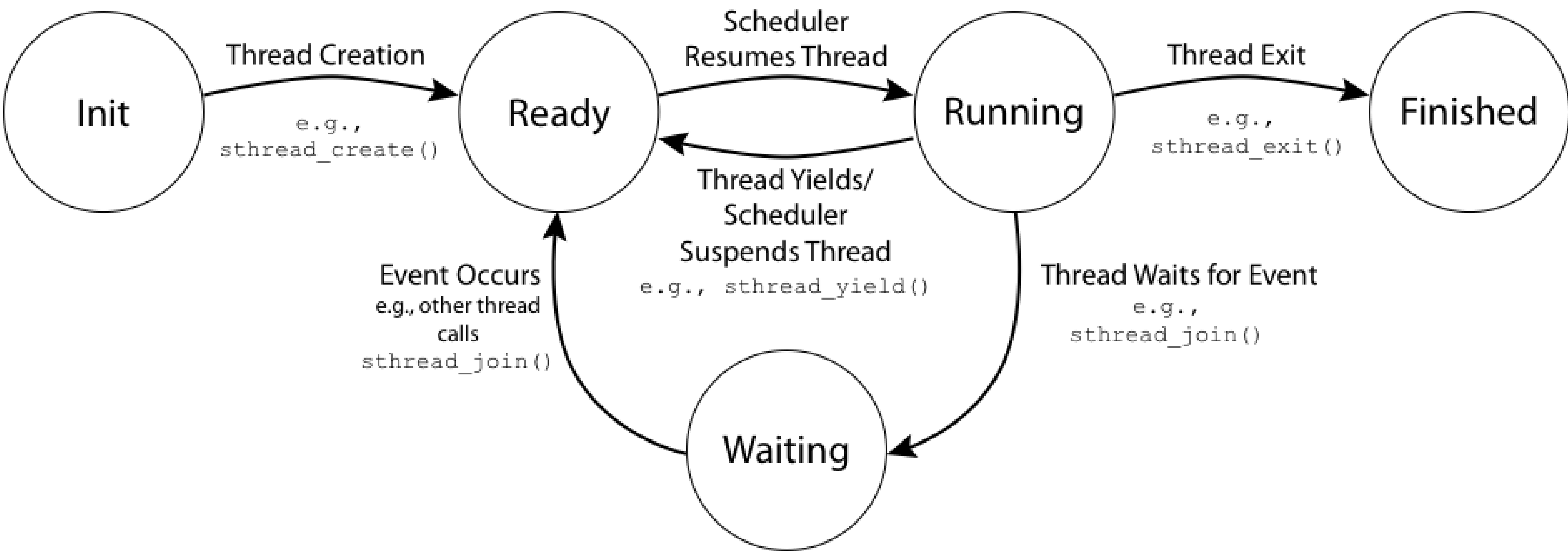
fork/join implementation of bzero

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

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

  for (i=0; i<NTHREADS; i++) {
    thread_join(threads[i]);
  }
}
```

Thread State



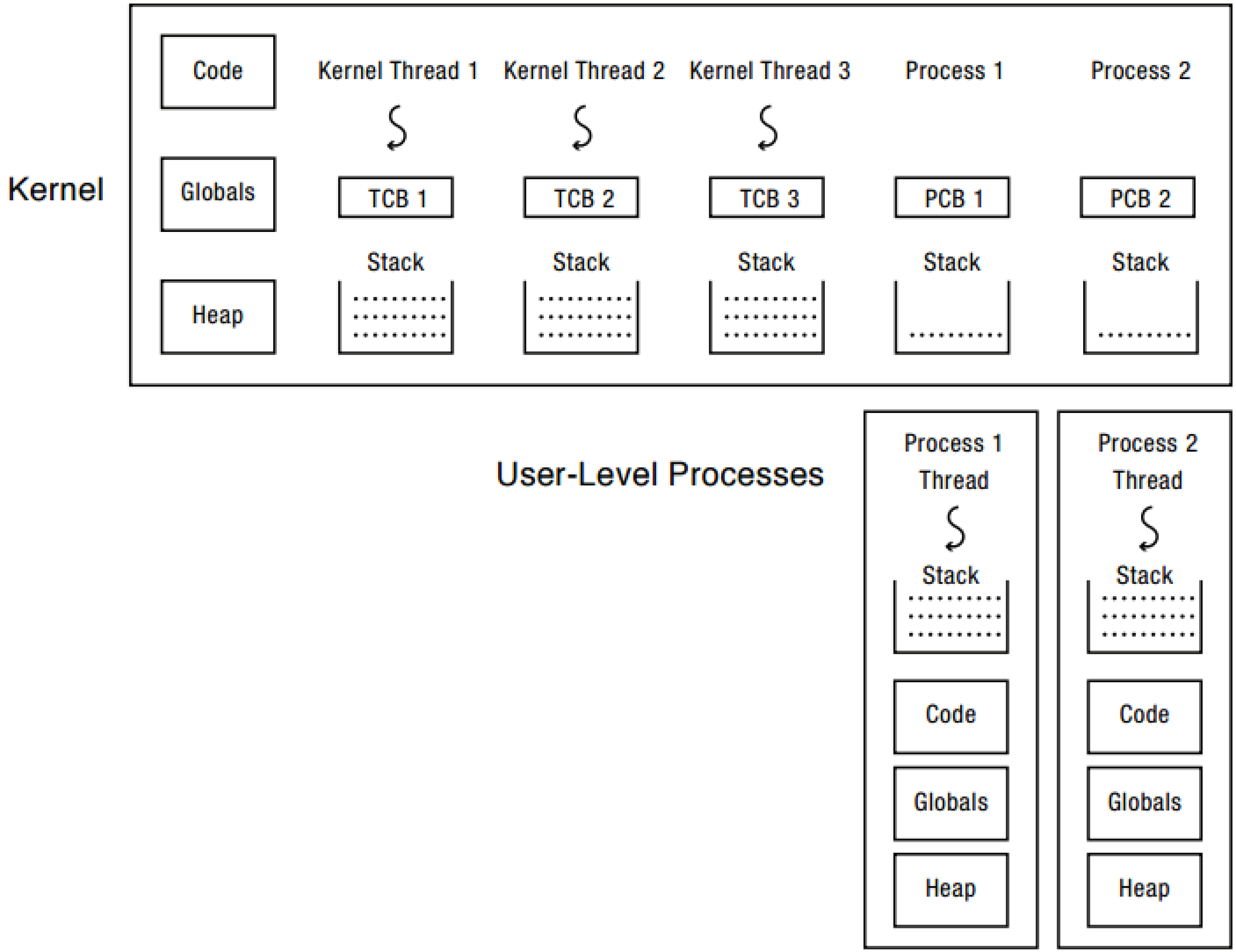
Thread Lifecycle



Implementing threads: Roadmap

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

Multithreaded OS Kernel



Implementing threads

- Thread_create(thread, 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): OS/161 mips_threadstart
  - Call (*func)(args)
  - When returns, call thread_exit()

Thread Stack

- What if a thread puts too many procedures on its stack?
  - What should happen?
  - What happens in Java?
  - What happens in Linux?
  - What happens in Pintos?

Implementing thread context switch

- Voluntary
  - thread_yield
  - thread_join (if child is not done yet)
- Involuntary
  - Interrupt or exception
  - Some other thread is higher priority

- *preemptive* vs. *non-preemptive* scheduling

Voluntary thread context switch

- User-level threads in a single-threaded process
  - Save registers on old stack
  - Switch to new stack, new thread
  - Restore registers from new stack
  - Return
- Kernel threads
  - Exactly the same!
  - OS/161: thread switch is always between kernel threads, not between user process and kernel thread

OS/161 switchframe_switch

```
/* a0: old thread stack pointer
 * a1: new thread stack pointer */

/* Allocate stack space for 10 registers. */
  addi sp, sp, -40

  /* Save the registers */
  sw   ra, 36(sp)
  sw   gp, 32(sp)
  sw   s8, 28(sp)
  sw   s6, 24(sp)
  sw   s5, 20(sp)
  sw   s4, 16(sp)
  sw   s3, 12(sp)
  sw   s2, 8(sp)
  sw   s1, 4(sp)
  sw   s0, 0(sp)

  /* Store old stack pointer in old thread */
  sw   sp, 0(a0)

/* Get new stack pointer from new thread */
  lw   sp, 0(a1)
  nop        /* delay slot for load */

/* Now, restore the registers */
  lw   s0, 0(sp)
  lw   s1, 4(sp)
  lw   s2, 8(sp)
  lw   s3, 12(sp)
  lw   s4, 16(sp)
  lw   s5, 20(sp)
  lw   s6, 24(sp)
  lw   s8, 28(sp)
  lw   gp, 32(sp)
  lw   ra, 36(sp)
  nop             /* delay slot for load */

  /* and return. */
  j ra
  addi sp, sp, 40     /* in delay slot */
```

x86 switch_threads(oldT, nextT)

```
# Save caller's register state
#  NOTE: %eax, etc. are ephemeral
pushl %ebx
pushl %ebp
pushl %esi
pushl %edi

# Get offsetof (struct thread, stack)
mov thread_stack_ofs, %edx
# Save current stack pointer to old
     thread's stack, if any.
movl SWITCH_CUR(%esp), %eax
movl %esp, (%eax,%edx,1)
```

```
# Change stack pointer to new
     thread's stack
# this also changes currentThread
movl SWITCH_NEXT(%esp), %ecx
movl (%ecx,%edx,1), %esp

# Restore caller's register state.
popl %edi
popl %esi
popl %ebp
popl %ebx
ret
```

A Subtlety

- thread_create(func) puts thread on ready list
- When it first runs, some thread calls switchframe
  - Saves old thread to stack
  - Restores next thread state from stack
- Set up a new thread's stack as if it had saved its state in switchframe
  - "returns" to stub at base of stack to run func

Two threads call yield

Thread 1's instructions	Thread 2's instructions	Processor's instructions
call thread_yield		call thread_yield
save state to stack		save state to stack
save state to TCB		save state to TCB
choose another thread		choose another thread
load other thread state		load other thread state
	call thread_yield	call thread_yield
	save state to stack	save state to stack
	save state to TCB	save state to TCB
	choose another thread	choose another thread
	load other thread state	load other thread state
return thread_yield		return thread_yield
call thread_yield		call thread_yield
save state to stack		save state to stack
save state to TCB		save state to TCB
choose another thread		choose another thread
load other thread state		load other thread state
	return thread_yield	return thread_yield
	call thread_yield	call thread_yield
	save state to stack	save state to stack
	save state to TCB	save state to TCB
	choose another thread	choose another thread
	load other thread state	load other thread state
return thread_yield		return thread_yield
...	...	...

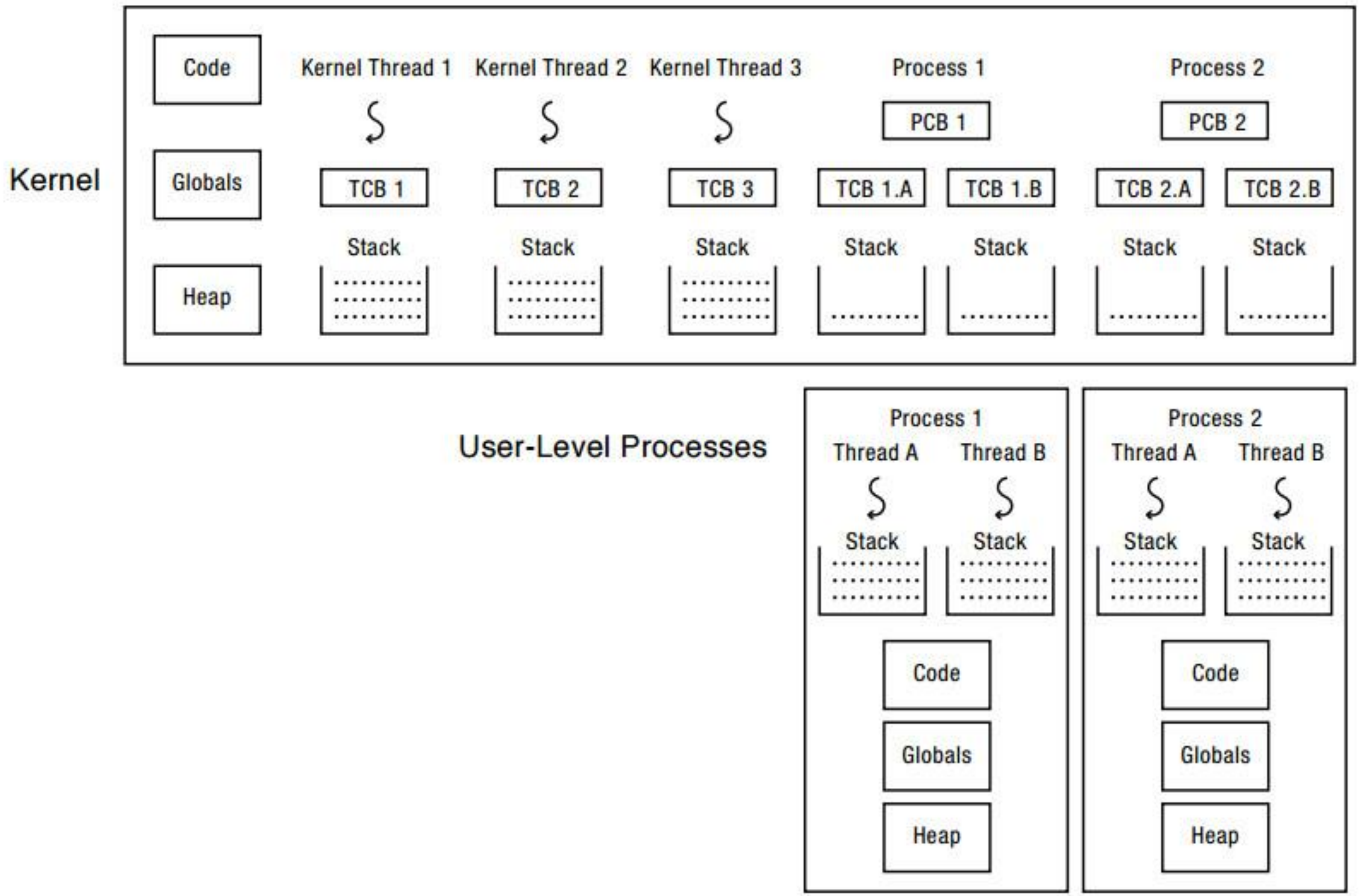
Involuntary thread switch

- Timer or I/O interrupt
  - Tells OS some other thread should run
- Simple version (OS/161)
  - End of interrupt handler calls schedule()
  - When resumed, return from handler resumes kernel thread or user process
- Faster version (Linux)
  - Interrupt handler returns to saved state in TCB
  - Could be kernel thread or user process

Multithreaded User Processes (Take 1)

- User thread = kernel thread (Linux, MacOS)
  - System calls for thread_create, thread_join, etc.
  - Kernel does context switch
  - Simple, but lots of transitions between user and kernel mode

Multithreaded User Processes (Take 1)



Multithreaded User Processes (Take 2)

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