

# Concurrency

# Motivation

- Operating systems (and application programs) often need to be able to handle multiple things happening at the same time
  - Process execution, interrupts, background tasks, system maintenance
- Humans are not very good at keeping track of multiple things happening simultaneously
- Threads are an abstraction to help bridge this gap

# 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

# 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?

# Definitions

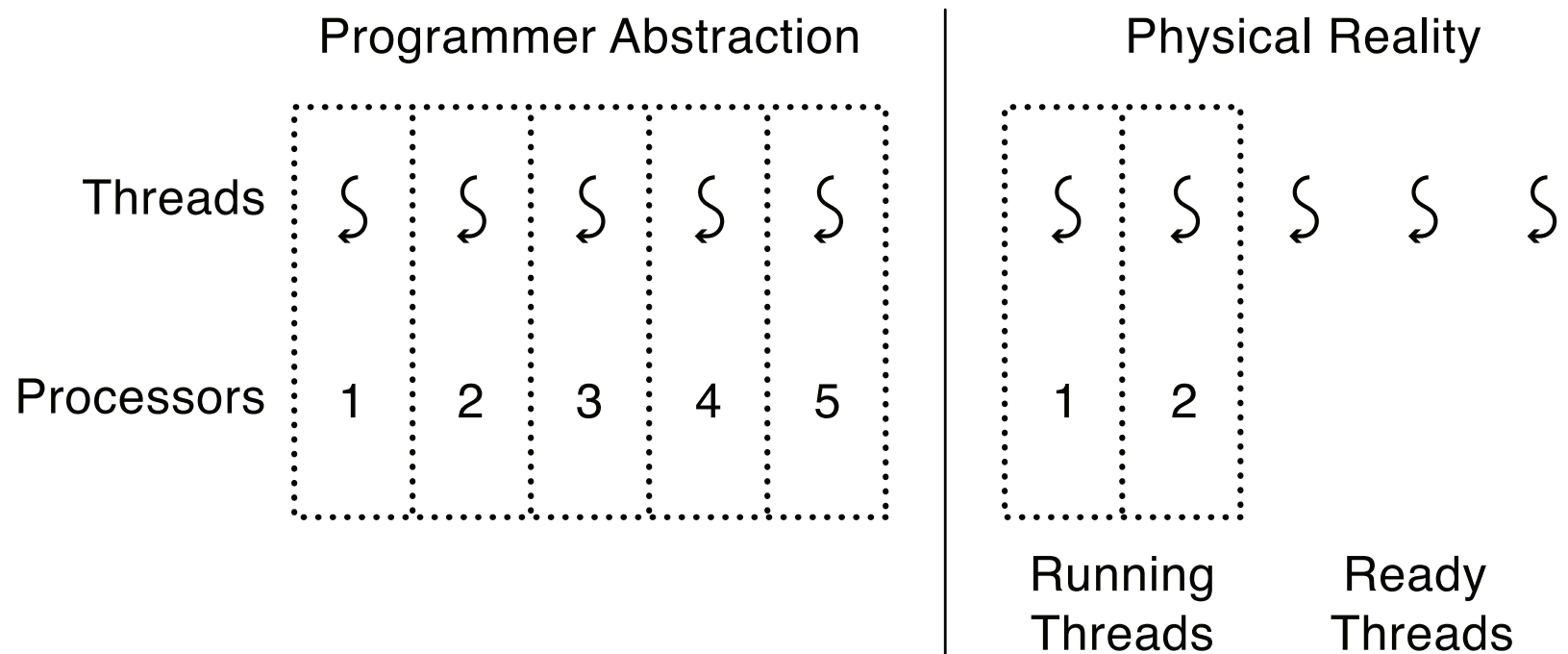
- A thread is a single execution sequence that represents a separately schedulable task
  - Single execution sequence: familiar programming model
  - Separately schedulable: OS can run or suspend a thread at any time
- Protection is an orthogonal concept
  - Can have one or many threads per protection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

# Thread Abstraction

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



# 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

```

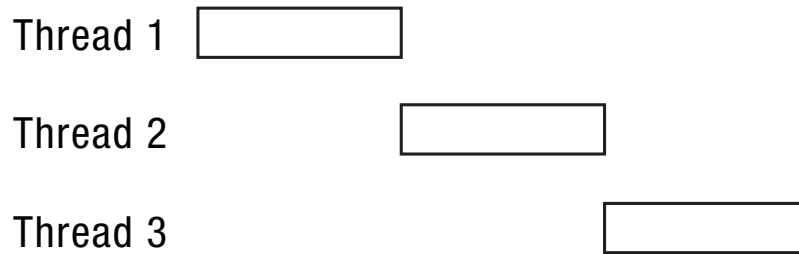
.
.
.
x = x + 1;
y = y + x;
.....
Thread is suspended.
Other thread(s) run.
Thread is resumed.
.....
z = x + 5y;

```

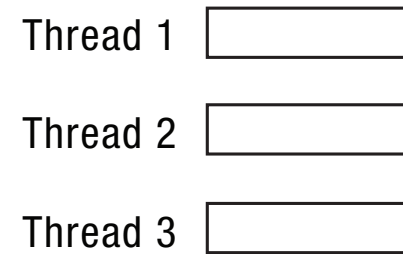


# Possible Executions

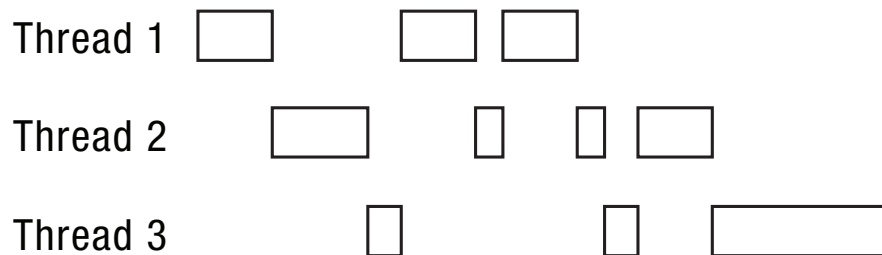
## One Execution



## Another Execution



## Another Execution



# 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

```
#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?
}
```

# threadHello: Example Output

- Why must “thread returned” print in order?
- What is maximum # of threads running when thread 5 prints hello?
- Minimum?

```
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.
```

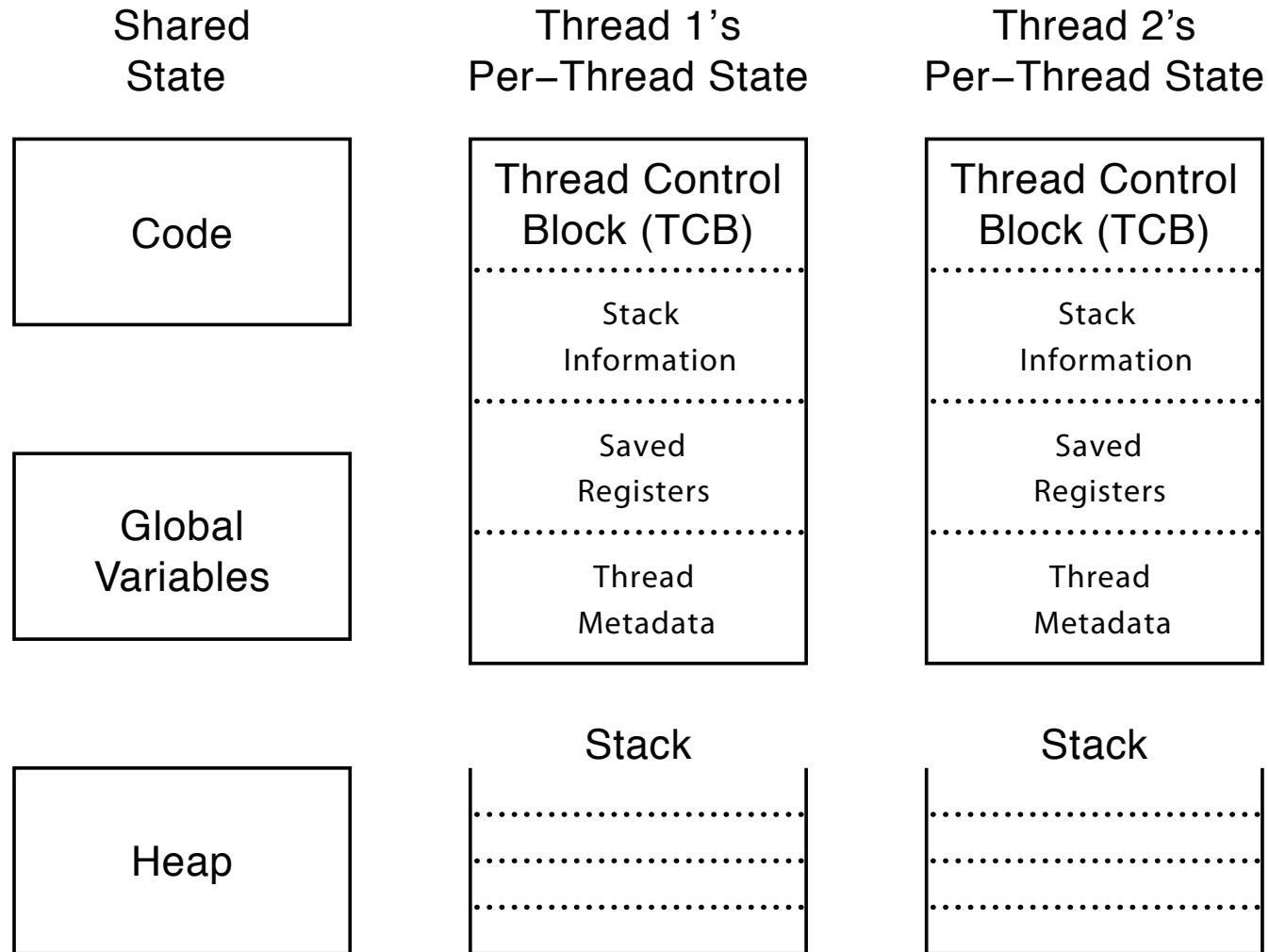
# Fork/Join Concurrency

- 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

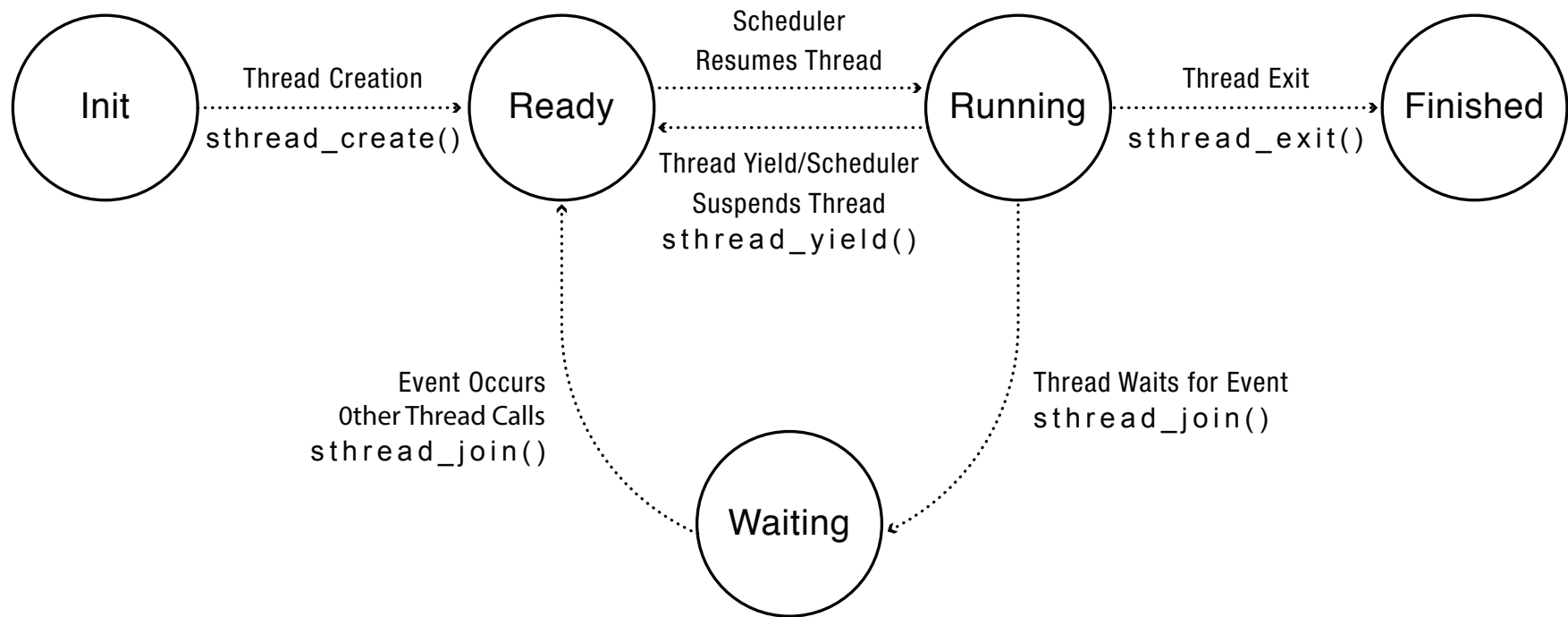
# bzero with fork/join concurrency

```
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

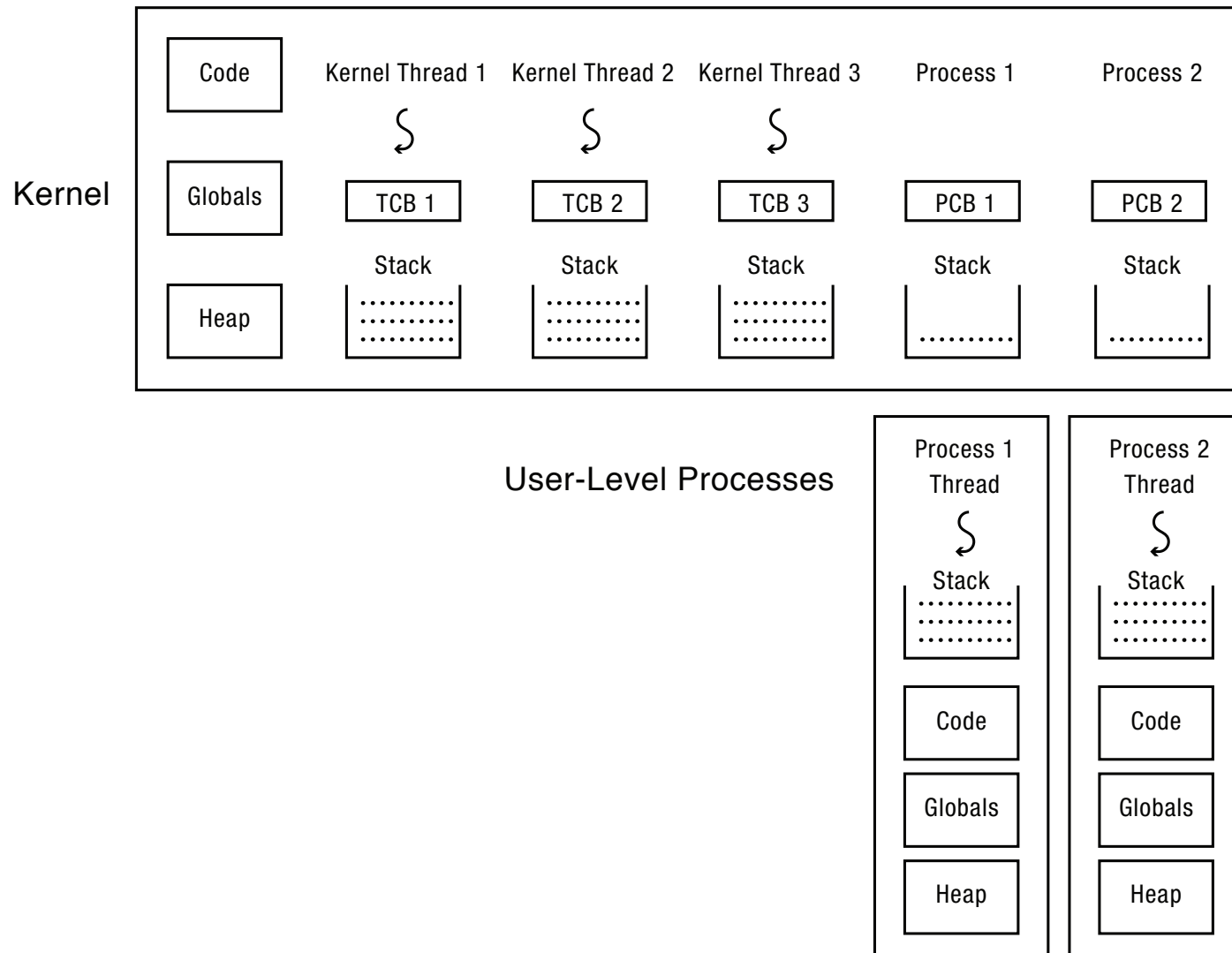




# Implementing Threads: Roadmap

- 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 (Linux, MacOS)
  - 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): OS/161 mips\_threadstart
  - Call (\*func)(args)
  - If return, call thread\_exit()

# Thread Stack

- What if a thread puts too many procedures on its stack?
  - What happens in Java?
  - What happens in the Linux kernel?
  - What happens in OS/161?
  - What *should* happen?

# Thread Context Switch

- Voluntary
  - Thread\_yield
  - Thread\_join (if child is not done yet)
- Involuntary
  - Interrupt or exception
  - Some other thread is higher priority

# Voluntary thread context switch

- Save registers on old stack
- Switch to new stack, new thread
- Restore registers from new stack
- Return
- Exactly the same with kernel threads or user threads
  - 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 offset of (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 puts new thread on ready list
- When it first runs, some thread calls switchframe
  - Saves old thread state to stack
  - Restores new thread state from stack
- Set up 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

"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

# Involuntary Thread Switch

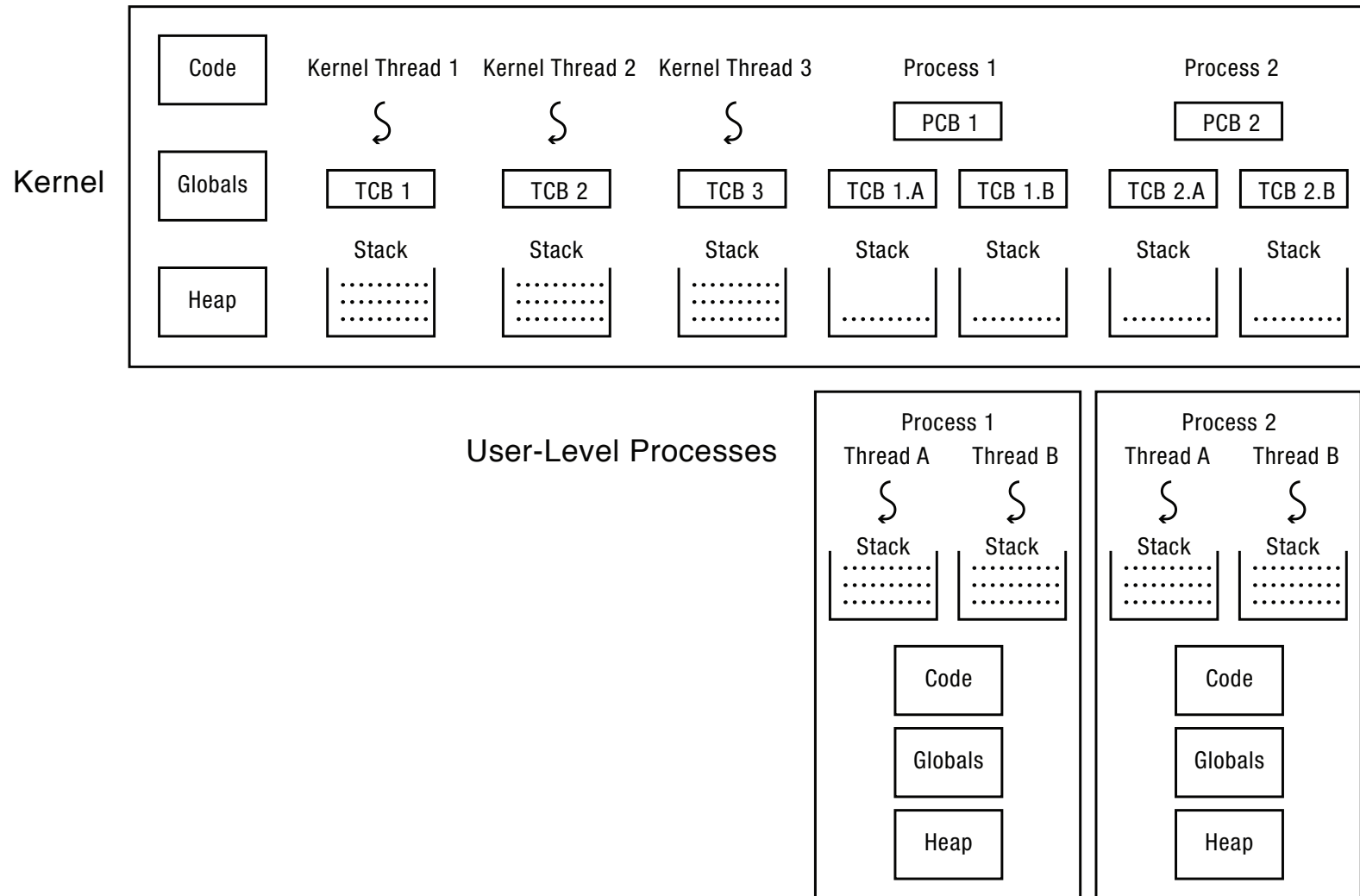
- Timer or I/O interrupt
  - Tells OS some other thread should run
- Simple version (OS/161)
  - End of interrupt handler calls `schedule()`
  - When resumed, return from handler resumes kernel thread or user process
- Faster version (Linux)
  - Interrupt handler returns to saved state in TCB
  - Could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

# Multithreaded User Processes

## (Take 1)

- User thread = kernel thread (Linux, MacOS)
  - System calls for thread fork, join, exit (and lock, unlock,...)
  - Kernel does context switch
  - Simple, but a lot 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/UNIX signal on timer interrupt
  - Use multiple processes for parallelism
    - Shared memory region mapped into each process

# Multithreaded 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 action requires (user-level) scheduling decision
  - Process assigned a new processor
  - Processor removed from process
  - System call blocks in kernel

# Question

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