Concurrency

*Throughout the course we will use overheads that were adapted from those distributed from the textbook website. Slides are from the book authors, modified and selected by Jean Mayo, Shuai Wang and C-K Shene.

An algorithm must be seen to be believed.

Donald Knuth

Spring 2019

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; but we do things concurrently very frequently.
- 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

Definitions

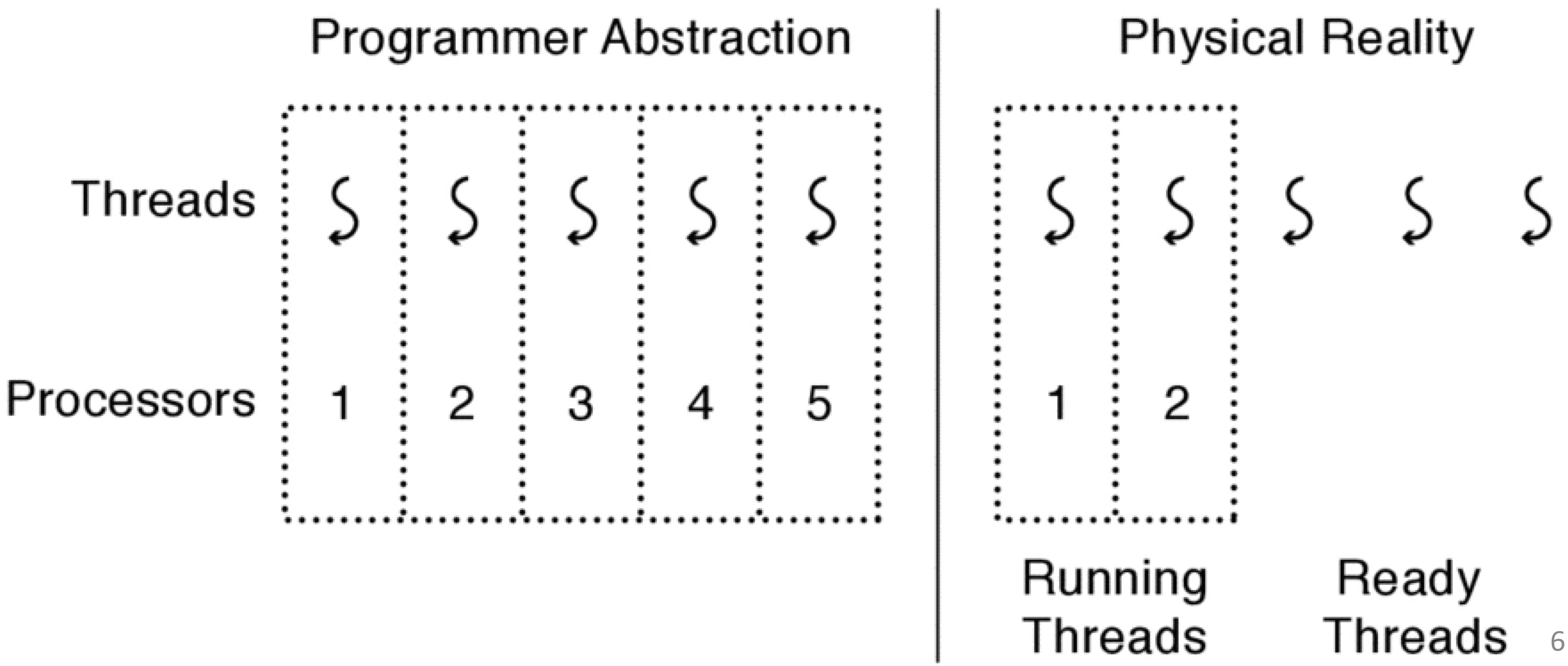
- A thread is a single execution sequence that represents a separately schedulable task
  - Single execution sequence (Sequential): familiar programming model
  - Separately schedulable: OS can run or suspend a thread at any time
- Protection is an orthogonal concept
  - Can have one or many threads 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
- **Multiprocess Kernel**
  - Multiple single-threaded processes
  - System calls access shared kernel data structures
- **Multiple Multi-Threaded User Processes**
  - Each with multiple threads, sharing same data structures, isolated from other user 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	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;

higher-level language statements are not atomic

But, situation can be worse, because higher-level statements are not atomic.
Each higher-level statement is translated to machine instruction,
and interrupt can happen between two instructions.

Machine Instruction View

Programmer's View	Machine's View
. . . . .	. . . . .

`x = x + 1`	`LOAD x`
	`ADD #1`
	`SAVE x`

`y = y + x`	`LOAD y`
	`ADD x`
	`SAVE y`

. . . . .	

Thread 1	Thread 2
`LOAD x`	
	`LOAD x`
`ADD #1`	
`SAVE x`	`ADD #1`
	`SAVE x`

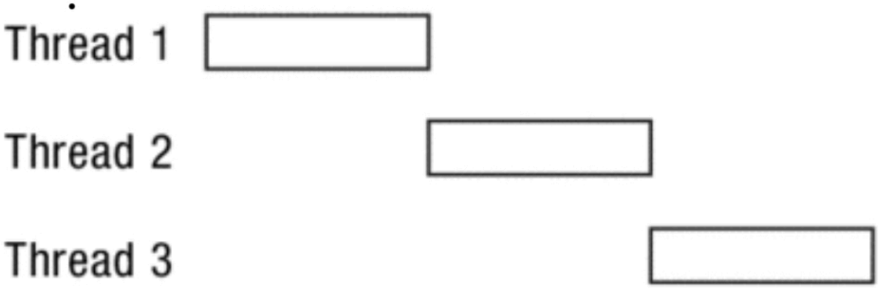
. . . .

what is the value of x here?

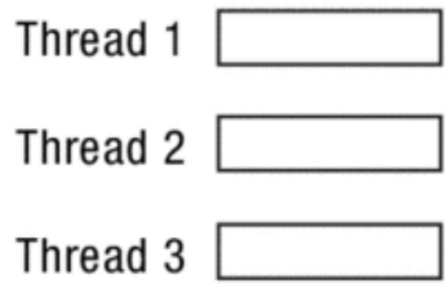
But, situation can be worse, because higher-level statements are not atomic.
Each higher-level statement is translated to machine instruction,
and interrupt can happen between two instructions.

Possible Executions

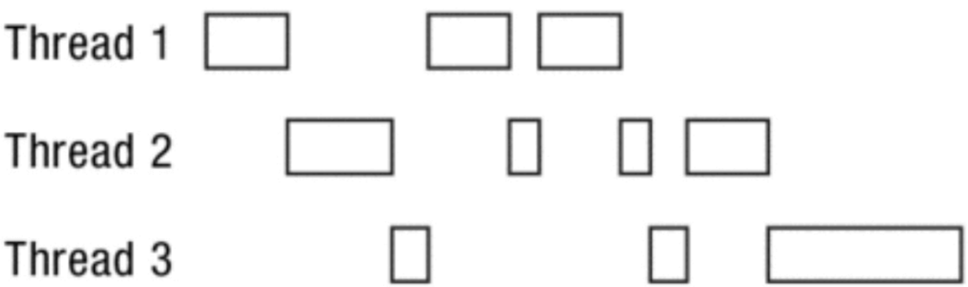
One Execution



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 (we do not do this assignment)**
- `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);
}
```

threadHello: Example Output

- Why must "thread returned" print in order?
- What is maximum # of threads running when thread 5 prints hello?
- Minimum?
- Are you certain about your answer?

```
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

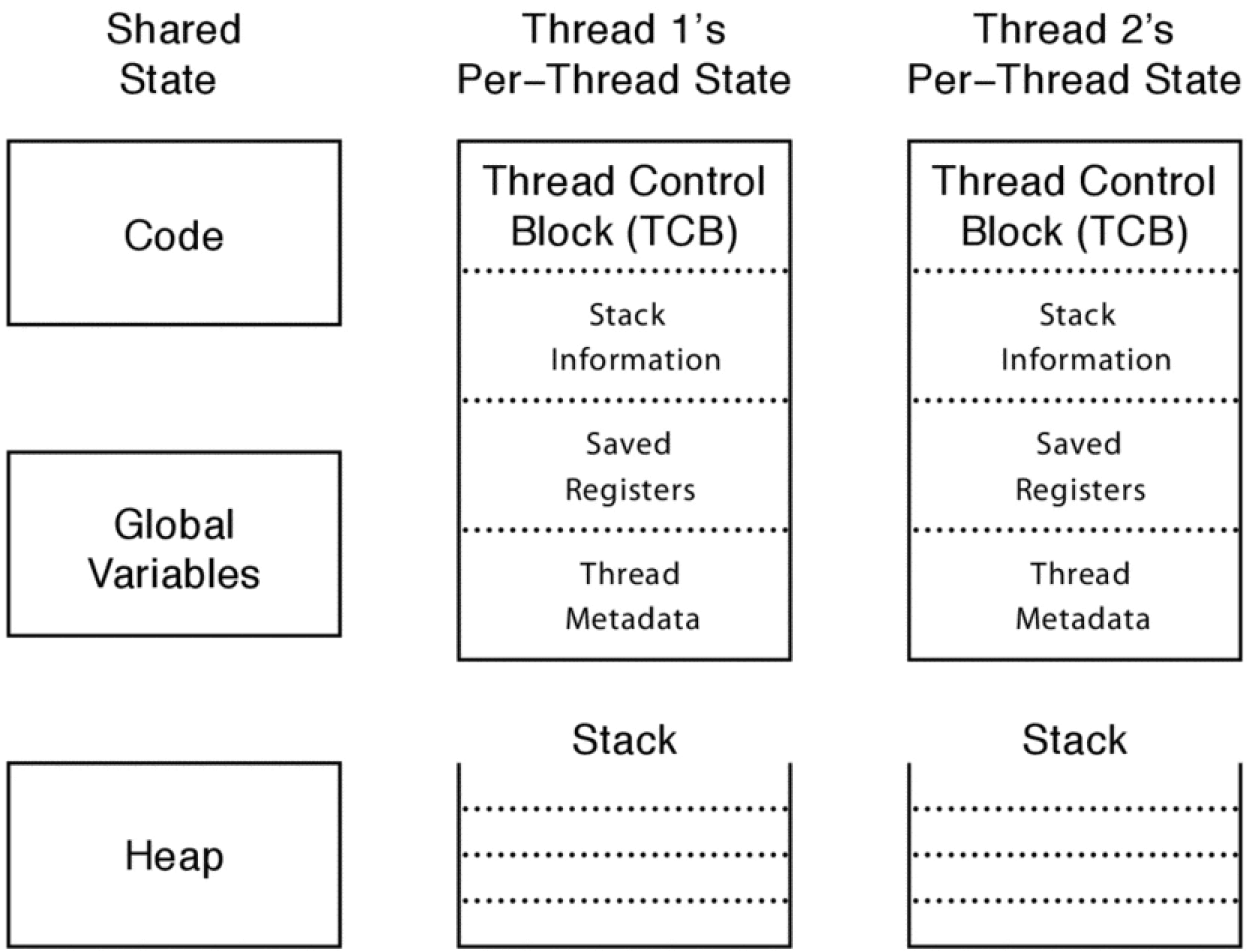
- Threads can create children, and wait for their completion
- Data only shared before fork/after join
- Examples:
  - Web server: fork a new thread for every new connection
    - As long as the threads are completely independent
  - Merge sort
  - Parallel memory copy

bzero with fork/join Concurrency

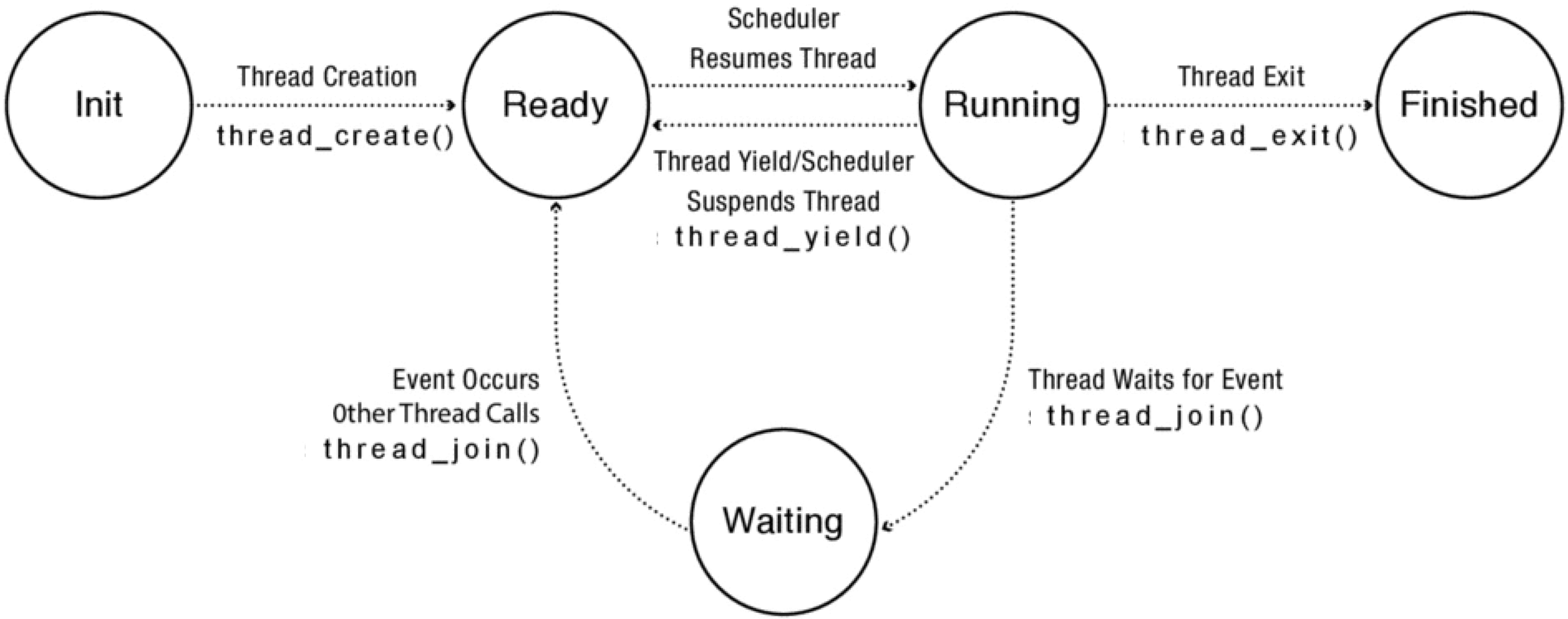
```
void blockzero (unsigned char *p, int length)
{
    int i, j;
    thread_t threads[NTHREADS];
    struct bzeroparams params[NTHREADS];

// 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(&(threads[i]), &go, &params[i]);
    }
    for (i = 0; i < NTHREADS; i++) {
        thread_join(threads[i]);
    }
}
```

Clear a block of memory to zero
Each thread clears a portion of the memory block

Thread Data Structures



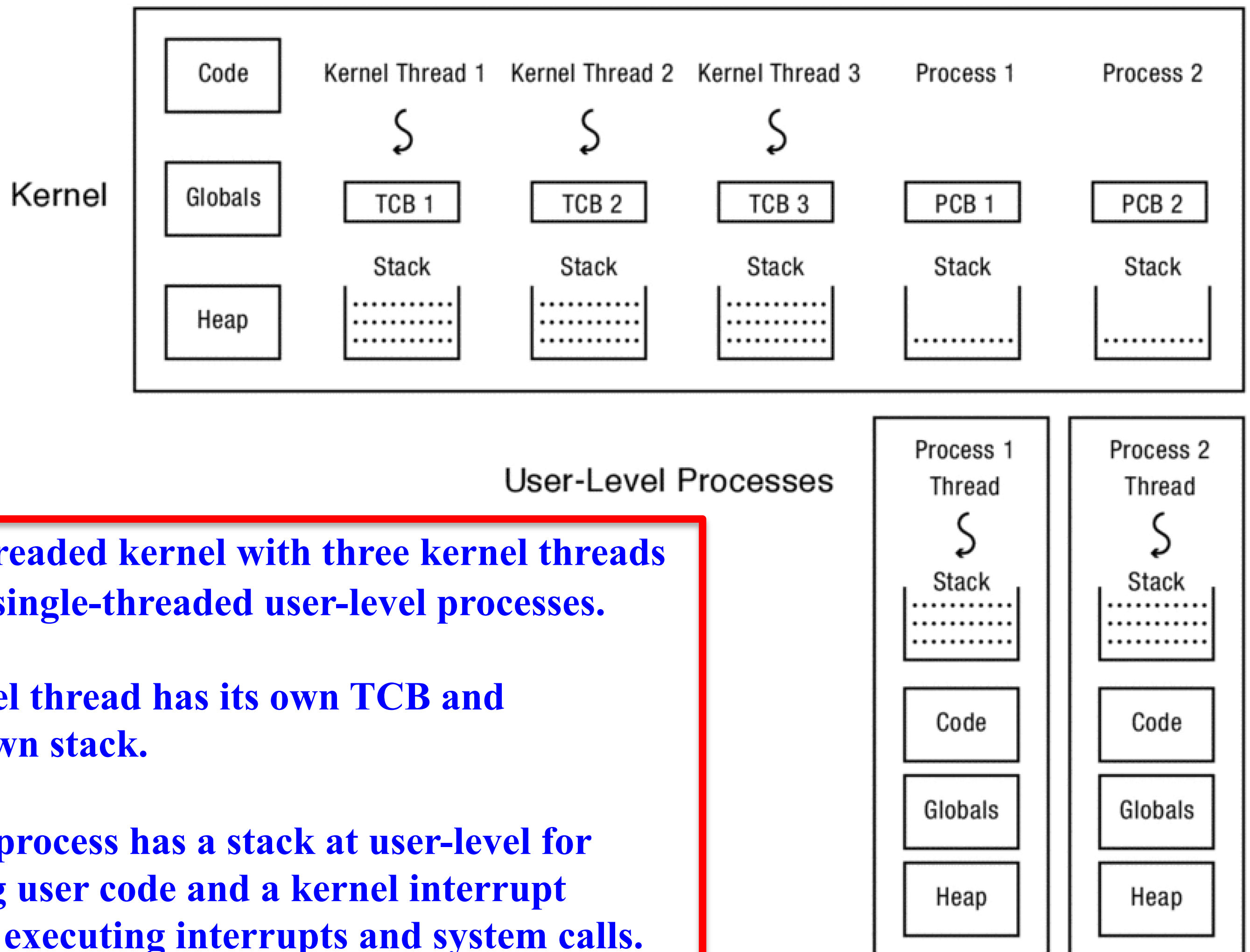
Thread Lifecycle



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

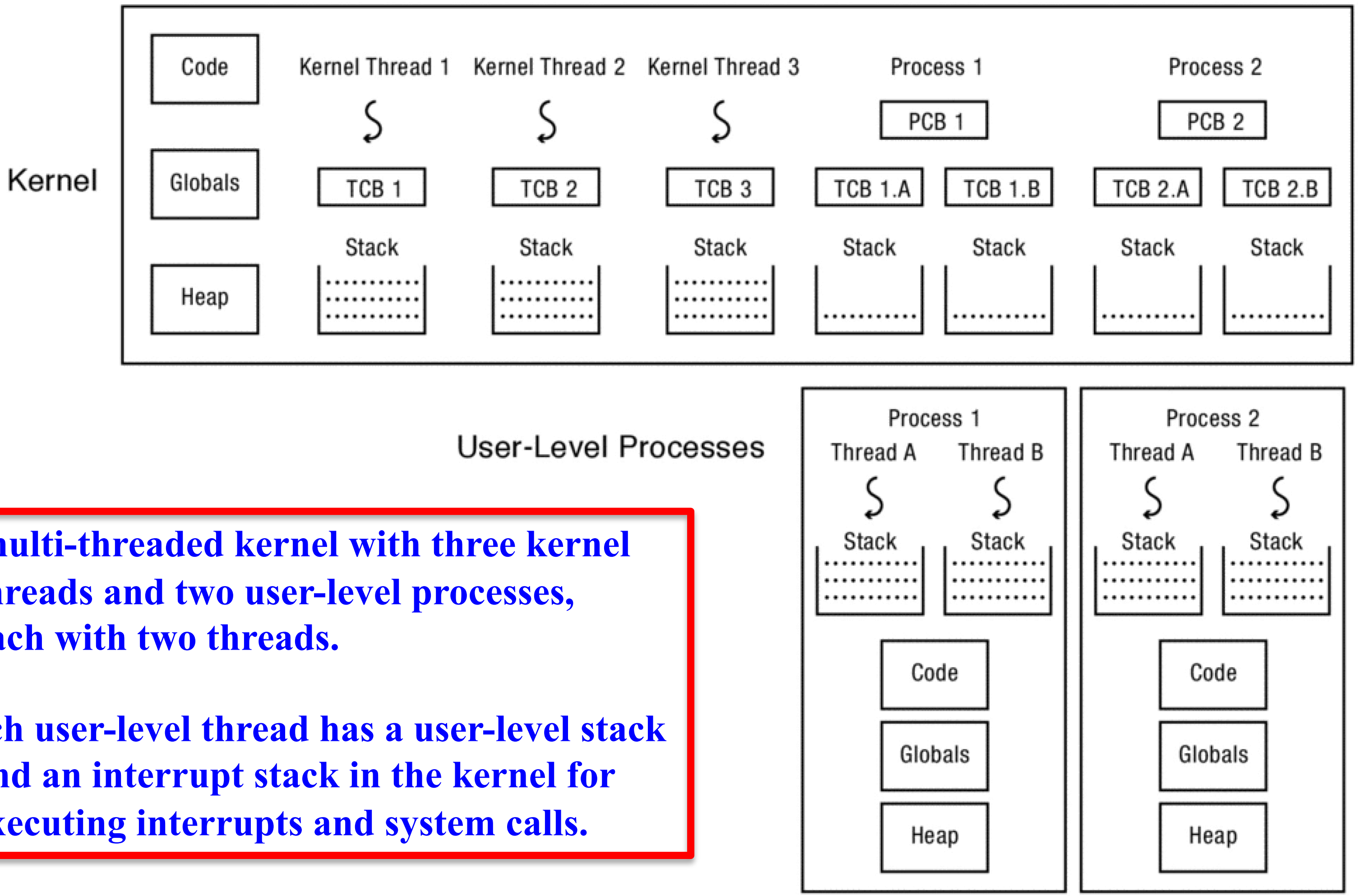


A multi-threaded kernel with three kernel threads and two single-threaded user-level processes.

Each kernel thread has its own TCB and and its own stack.

Each user process has a stack at user-level for executing user code and a kernel interrupt stack for executing interrupts and system calls.

Multithreaded User Processes



A multi-threaded kernel with three kernel threads and two user-level processes, each with two threads.

Each user-level thread has a user-level stack and an interrupt stack in the kernel for executing interrupts and system calls.

Implementing Threads

- `thread_create(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()`

Thread Creation (Pseudo-Code)

```
thread_create(thread_t *thread, void (*func)(int), int arg)
{
   TCB *tcb = new TCB();                          // allocate TCB and stack

   thread->tcb = tcb;
   tcb->stack_size = INITIAL_STACK_SIZE;
   tcb->stack = new Stack(INITIAL_STACK_SIZE);

   tcb->sp = tcb->stack + INITIAL_STACK_SIZE;  // initialize registers
   tcb->pc = stub;

   *(tcb->sp) = arg;        // push the argument and function on to stack
   tcb->sp--;
   *(tcb->sp) = func;
   tcb->sp--;

   thread_dummySwitchFrame(tcb);                 // to be discussed later
   tcb->state = READY;
   readyList.add(tcb);
}

void stub(void (*func)(int), int arg)
{
   (*func)(arg);          // execute the function func()
   thread_exit(0);        // if func() does not call exit, call it here
}
```

Thread Deletion: 1/3

- Two steps are needed to delete the thread when `thread_exit` is called.
  - Remove the thread from the ready list so that it will never run again.
  - Free the per-thread state allocated for the thread.
  - **What if an interrupt occurs before the thread finishes de-allocating its state → memory leak!**

Thread Deletion: 2/3

- Who is responsible to free a thread from its state? The thread itself?
  - **If the thread frees its state, it does not have a stack to finish its code in `thread_exit`.**
  - **What if an interrupt occurs just after the running thread's stack has been de-allocated?** If the context switch code tries to save the current's state, it will be writing to de-allocated memory, which may have been allocated to other thread for some other data structure.

Thread Deletion: 3/3

- **Solution**:
  - Freeing a thread's state has to be done by some other thread. On exit, the thread…
    1) transitions to the **FINISHED** state
    2) moves its TCB from the ready list to the finished list so that the scheduler will never run it.
    3) Once the finished thread is no longer running, it is safe for some other thread to free the state of the thread.

Thread Context Switch

- **Voluntary**
  - `thread_yield`
  - `thread_join` **(if child is not done yet)**
- **Involuntary**
  - **Interrupt or exception**
  - **Some other thread has 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

Thread Switch x86 (Pseudo-Code)

```
void thread_switch(oldThreadTCB, newThreadTCB)
{
   pushad;                       // push general register values
                                 //     onto the old stack
   oldThreadTCB->sp = %esp;      // save the old thread's SP
   %esp = newThreadTCB->sp;      // switch to the new stack
   popad;                        // pop register values from
                                 //     the new stack
   return;
}
```

This function enters as `oldThread`, but returns as `newThread`.
It returns with `newThread`'s registers and stack

Thread Yield x86 (Pseudo-Code)

```
void thread_yield()
{
   TCB  *chosenTCB, *finishedTCB;

   disableInterrupts();                    // disable interrupts
   chosenTCB = readyList.getNextThread();   // choose the next
   if (chosenTCB == NULL) {
               // nothing to run.  returns to the original thread
   }
   else {                  // move running thread onto the ready list
      runningThread->state = READY;
      readyList.add(runningThread);
      thread_switch(runningThread, chosenTCB);   // switch to new
      runningThread->state = running;
   }
                           // delete any threads on the finished list
   while (finishedTCB = finishedList->getNextThread()) != NULL) {
      delete finishedTCB->stack;
      delete finishedTCB;
   }
   enableInterrupts();
}
```

Thread Switch Frame x86

```
void thread_dummySwitchFrame(newThread)
{
    *(tcb->sp) = stub;
    tcb->sp—
    tcb->sp -= SizeOfPopad;
}
```

`thread_create` must put a dummy frame at the top of its stack: the return PC and space for `pushad` to have stored a copy of registers.

Therefore, when someone switches to a newly created thread, the last two lines of `thread_switch` work correctly.

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

```
# 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` **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**

Involuntary Thread/Process Switch

- Timer or I/O interrupt
  - Tells OS some other thread should run
- Simple version (OS/161)
  - End of interrupt handler calls `switch()`
  - When resumed, return from handler resumes kernel thread or user process
  - Thus, processor context is saved/restored twice (once by interrupt handler, once by thread switch)

Faster Thread/Process Switch

- What happens on a timer (or other) interrupt?
  - Interrupt handler saves state of interrupted thread
  - Decides to run a new thread
  - Throw away current state of interrupt handler!
  - Instead, set saved stack pointer to trapframe
  - Restore state of new thread
  - On resume, pops trapframe to restore interrupted thread

Multithreaded User Processes 1/3

- 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 2/3

- 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 3/3

- 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

The End