Ch 4: Concurrency and Threads

Bill Cheng

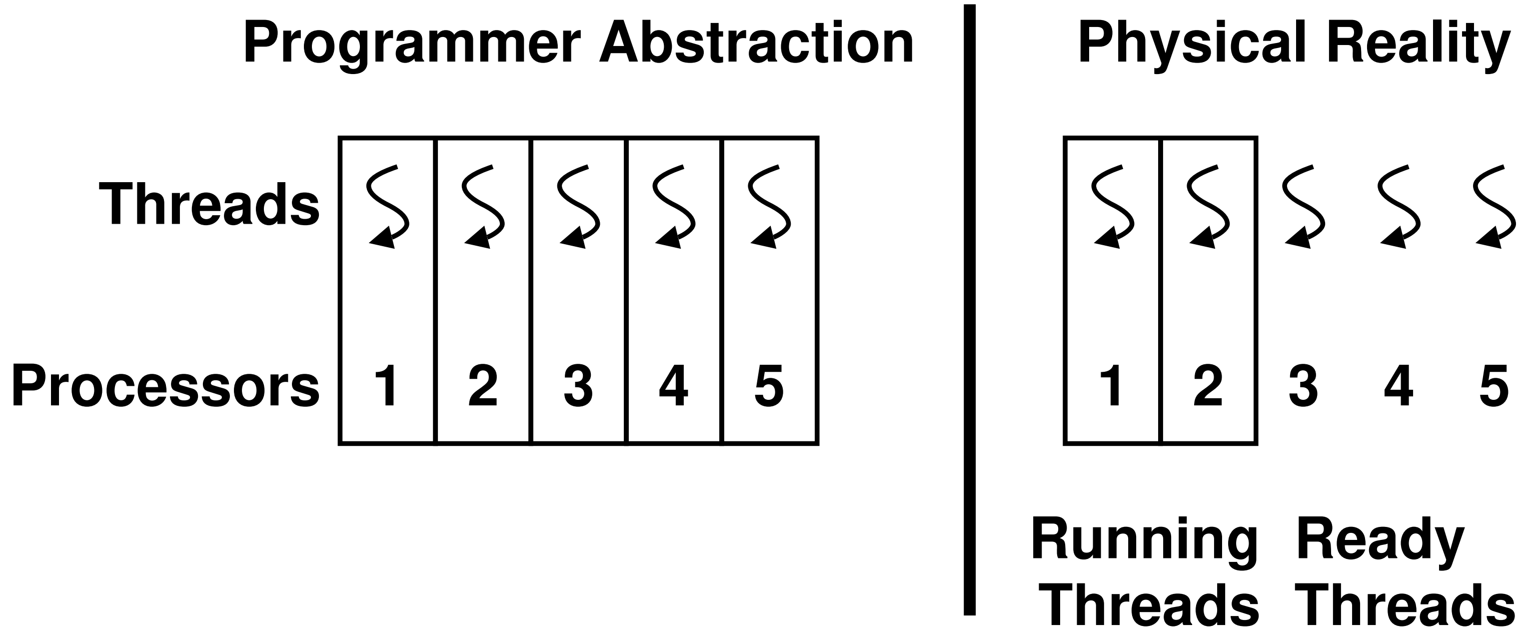
http://merlot.usc.edu/william/usc/

Concurrency

- Concurrency: multiple activities can happen at the same time
  - some would make a distinction between *concurrency* and *parallelism*
    - *parallelism* refers to hardware parallelism (i.e., true parallelism)
    - *concurrency* means to juggle things so fast that it looks like things are happening in parallel (but there is only one piece of hardware, and therefore, you cannot have true parallelism)
    - looks like our textbook does not make such a distinction
- Correctly managing concurrency is a key challenge for OS developers
  - need a structured approach to concurrency and we will learn how to do it

Concurrency

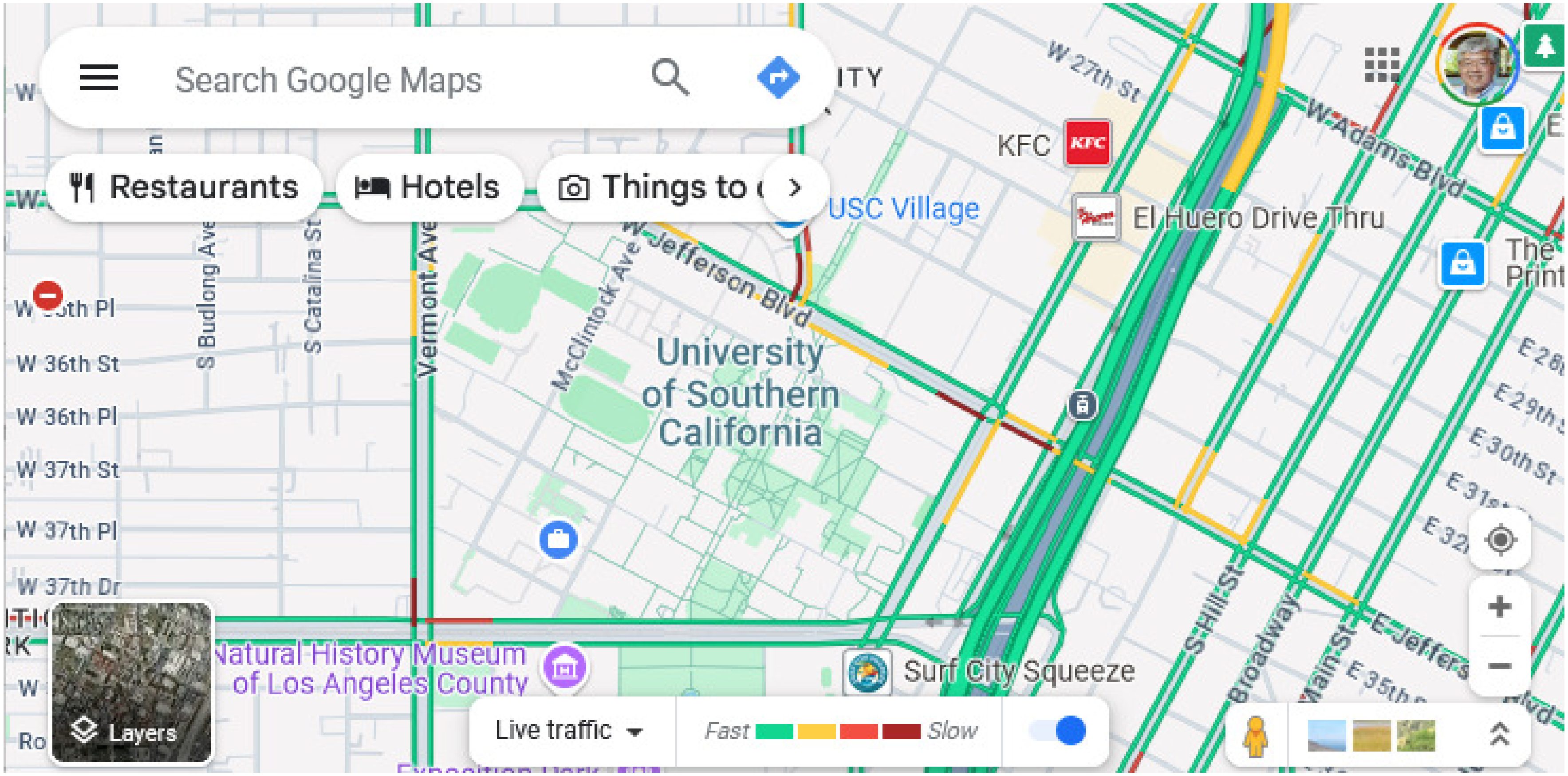
- Write concurrent program as a set of sequential streams of execution, called threads, that interact and share results in very precise ways
  - each thread behaves as if it has its own dedicated processor



- We need to learn to write multi-threaded programs so that no matter how many processors we have (from 1 to infinity), we will always get a correct result

(4.1) Thread Use Cases

Multi-threaded App: Google Map



- **Use many threads that runs concurrently**
  - **multiple threads for retrieving map data and drawing portions of the screen, one or more threads to handle UI widgets, threads to fetch advertisements and recommendations, etc.**

Four Reasons to Use Threads

- Advantages of using threads
  - *good way to structure program:* expressing logically concurrent tasks
    - programs often interact with or simulate real-world applications that have concurrent activities and threads let you express an application's natural concurrency by writing each concurrent task as a separate thread
  - *responsiveness:* shifting work to run in the background
    - use a separate thread to interact with the user can make the application feel like it's more responsive
    - OSes use threads to preserve responsiveness
      - OSes are designed so that common case is fast
  - *performance:* exploiting multiple processors
  - *performance:* managing I/O devices
    - hardware can run in parallel
    - when one thread is waiting for I/O device to complete, another thread can be using the processor

(4.2) Thread Abstraction

Thread

- A *thread* is a single execution sequence that represents a separately schedulable task
  - *single execution sequence:* the familiar programming model
  - *separately schedulable task:* the OS can run, suspend, or resume a thread at any time
    - therefore, an interrupt handler is not a thread

- To map an arbitrary set of threads to a fixed set of processors, operating systems include a *thread scheduler* that can switch between threads that are running and those that are *ready* but not running
  - threads that are ready to run (i.e., not waiting for anything else but only waiting for a processor to run on) are put on a *ready list*

- The abstraction is that each thread runs on a dedicated *virtual processor* with unpredictable and variable speed
  - the scheduler may suspend a thread between one instruction and the next and resume running it later

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

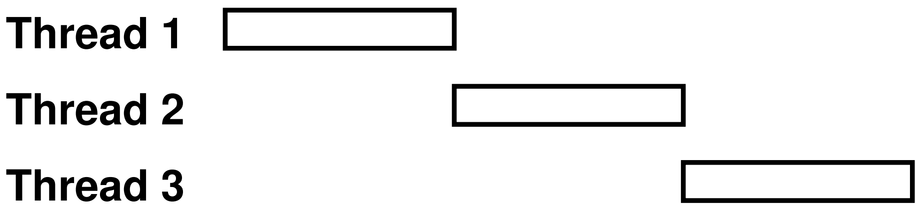
```
.
.
.
x = x + 1;
---------------
thread suspended
other threads run
thread is resumed
---------------
y = y + x;
z = x + 5y;
.
.
.
```

Possible Execution #3

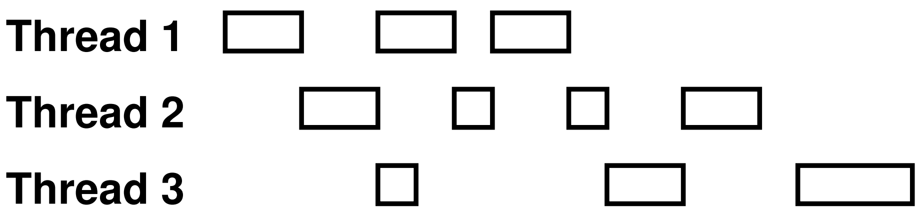
```
.
.
.
x = x + 1;
y = y + x;
---------------
thread suspended
other threads run
thread is resumed
---------------
z = x + 5y;
.
.
.
```

Possible Executions

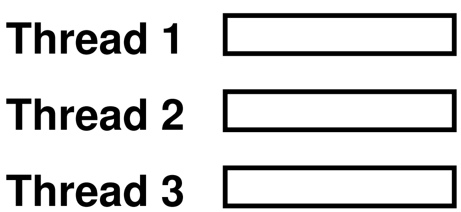
One Execution (1 CPU)



Another Execution (1 CPU)



Another Execution (3 CPUs)



Cooperative vs. Preemptive Multi-threading

- ***Cooperative multi-threading:*** a thread runs without interruption until it explicitly relinquishes control of the processor
  - DOS/Window 3.x, Macintosh
  - can only run one thread at a time, no parallelism
  - most people would not consider the OS they run on a "real" OS

- ***Preemptive multi-threading:*** running threads can be switched at any time
  - when we talk about multi-threading in this class, we mean ***preemptive multi-threading***

(4.3) Simple Thread API

Thread Operations

- `thread_create(thread, func, args)`
  - create a new thread to run `func(args)`
    - sometimes I would refer to `func` as the *first procedure* of the new thread
- `thread_yield()`
  - relinquish processor voluntarily
- `thread_join(thread)`
  - wait for thread to die
  - this function is a *blocking call* (i.e., the calling thread may get suspended and may not return right away)
  - may only be called once for each thread
- `thread_exit()`
  - quit thread and clean up, *wake up* joiner if any

Example: Multi-threaded Hello World

```
#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
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.
```

⇨ **After `thread_create()` returns successfully, when will the newly created thread starts running?**

Fork/Join Parallelism

- Threads can create children, and wait for their completion
- You can write code such that data only shared before fork/after join
  - therefore, can run the children in parallel
- Examples:
  - web server: "fork" a new thread for every new connection
    - as long as the threads are completely independent
  - merge sort: each level of merge sort can run in parallel
  - parallel memory copy

bzero() with Fork/Join Parallelism

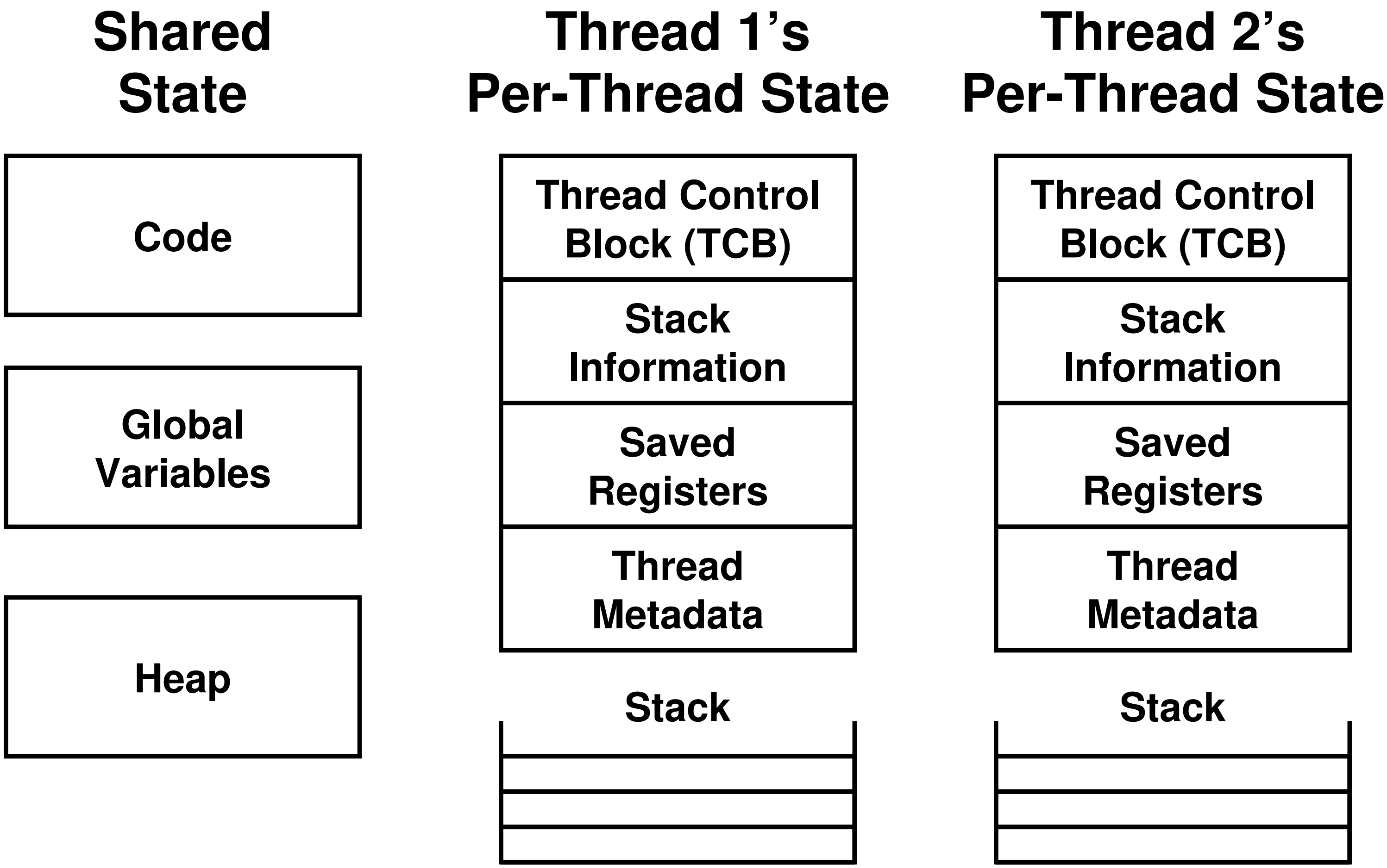
```
// to pass two arguments, we need a struct to hold them
typedef struct bzeroparams {
  unsigned char *buffer;
  int length;
};

#define NTHREADS 10
void go(struct bzeroparams *p) {
  memset(p->buffer, 0, p->length);
}

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]);
  }
}
```

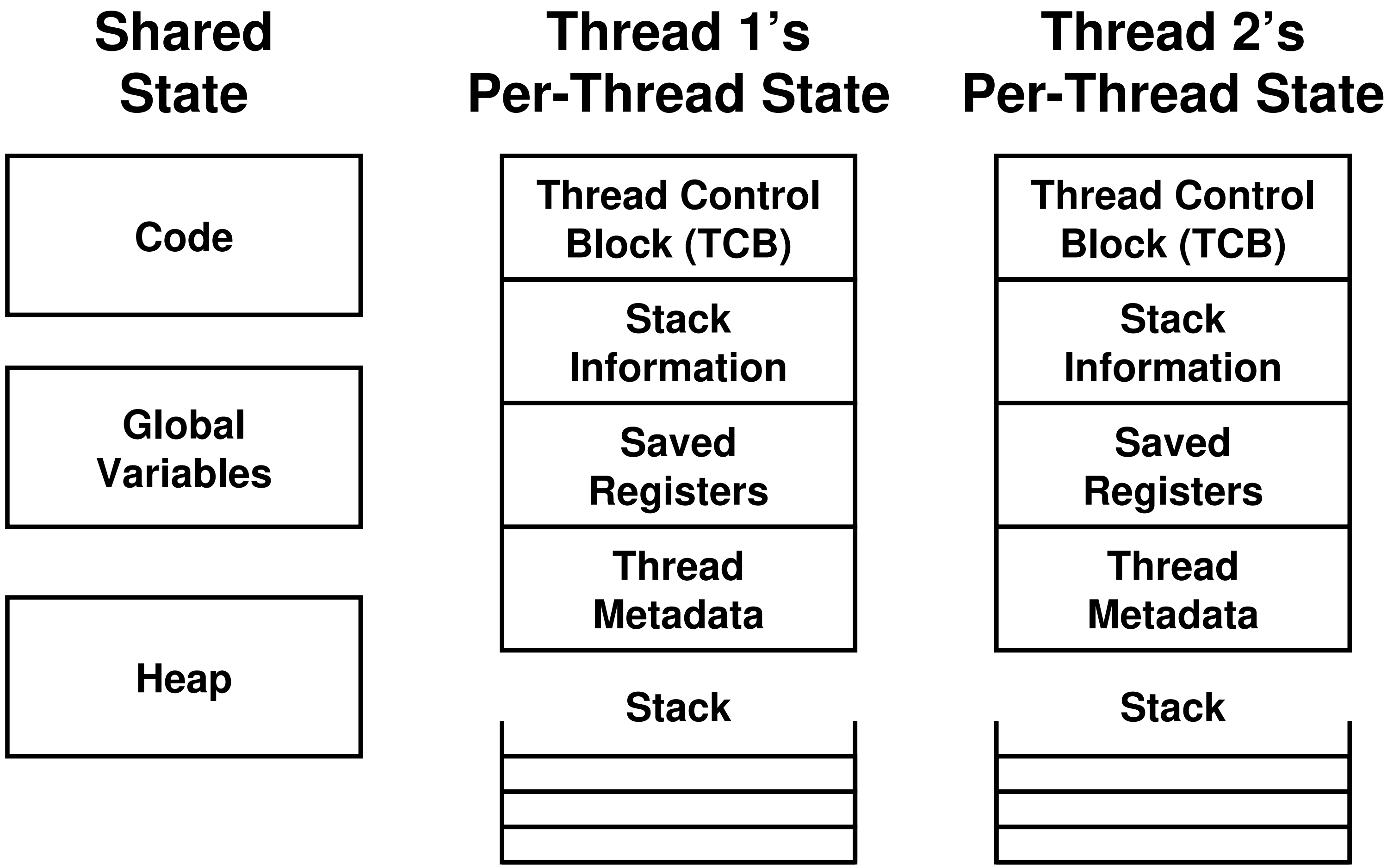
(4.4) Thread Data Structures and Life Cycle

Thread Data Structures



- Above can be a ***multi-threaded process*** or the ***OS kernel***
- Examples of thread metadata: thread ID, scheduling priority, owner, resource consumption, etc.

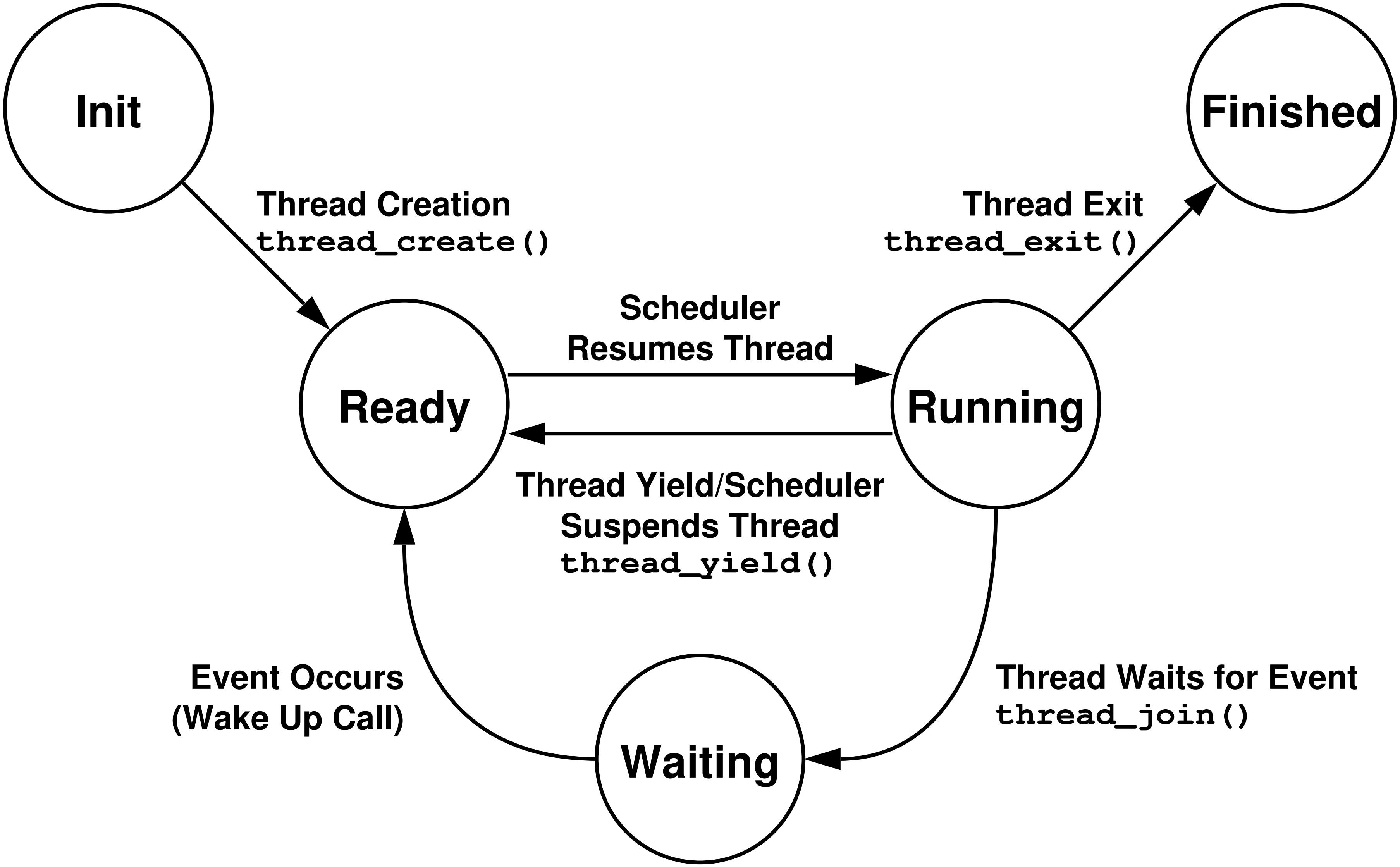
Thread Data Structures



- To avoid unexpected behaviors, it's important to know which variables are designed to be shared and which are designed to be private
  - e.g., `errno` is private to a thread

(4.5) Thread Life Cycle

Thread Lifecycle



- ***Wake up*** **means moving a thread from the queue it's sleeping on to the ready queue (and changing state)**

Thread Lifecycle

- *Ready* state (a.k.a., *runnable* state): thread is on the *ready list*
  - the ready list is conceptually a list, but usually, it can be a complex data structure to support whatever the scheduler needs
- *Running* state: thread is on a processor (and not on the ready list)
  - stored register values on the TCB are stale
  - a preemptive scheduler can preempt a running thread and move it to the ready state
  - a running thread can call `thread_yield()` to go back into the ready state
- *Waiting* state: thread is *sleeping on a queue* (a waiting list of a synchronization variable), waiting for an *wake up* event
  - when wake up event occurs, thread is moved to the ready state
- *Finished* state (a.k.a., *zombie* state): thread is dead and can never run code again
  - some state of the thread may be freed

Thread Lifecycle

Thread State	Location of TCB	Location of Registers
Init	(being created)	TCB
Ready	ready list	TCB
Running	running list	processor
Waiting	waiting list (for a synchronization variable)	TCB
Finished	Finished	TCB or deleted

Idle Thread

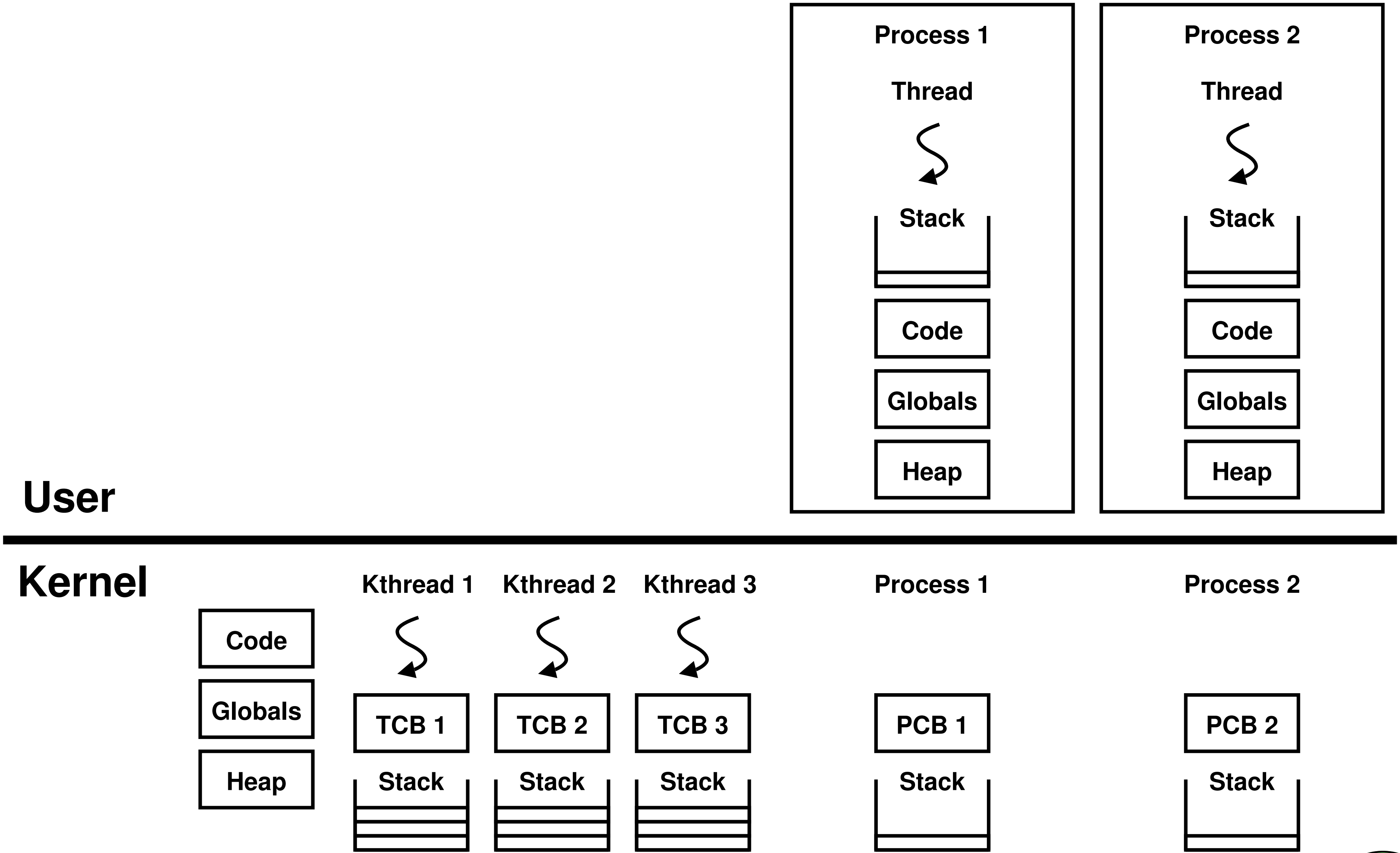
- On a multi-processor system, it may not be possible to keep all the processors busy
  - create a low priority *idle thread* for each processor
    - the idle thread sits in an infinite loop calling `thread_yield()`
    - to save power, idle thread would halt the CPU (i.e., put the processor into a low-power sleep mode)
      - halt only returns if the processor gets a hardware interrupt and this will get the idle thread going again
      - halting the CPU is a privileged instruction, if this is done inside a VM, the host OS can switch to a differernt VM

(4.6) Implementing Kernel Threads

Implementing Threads: Roadmap

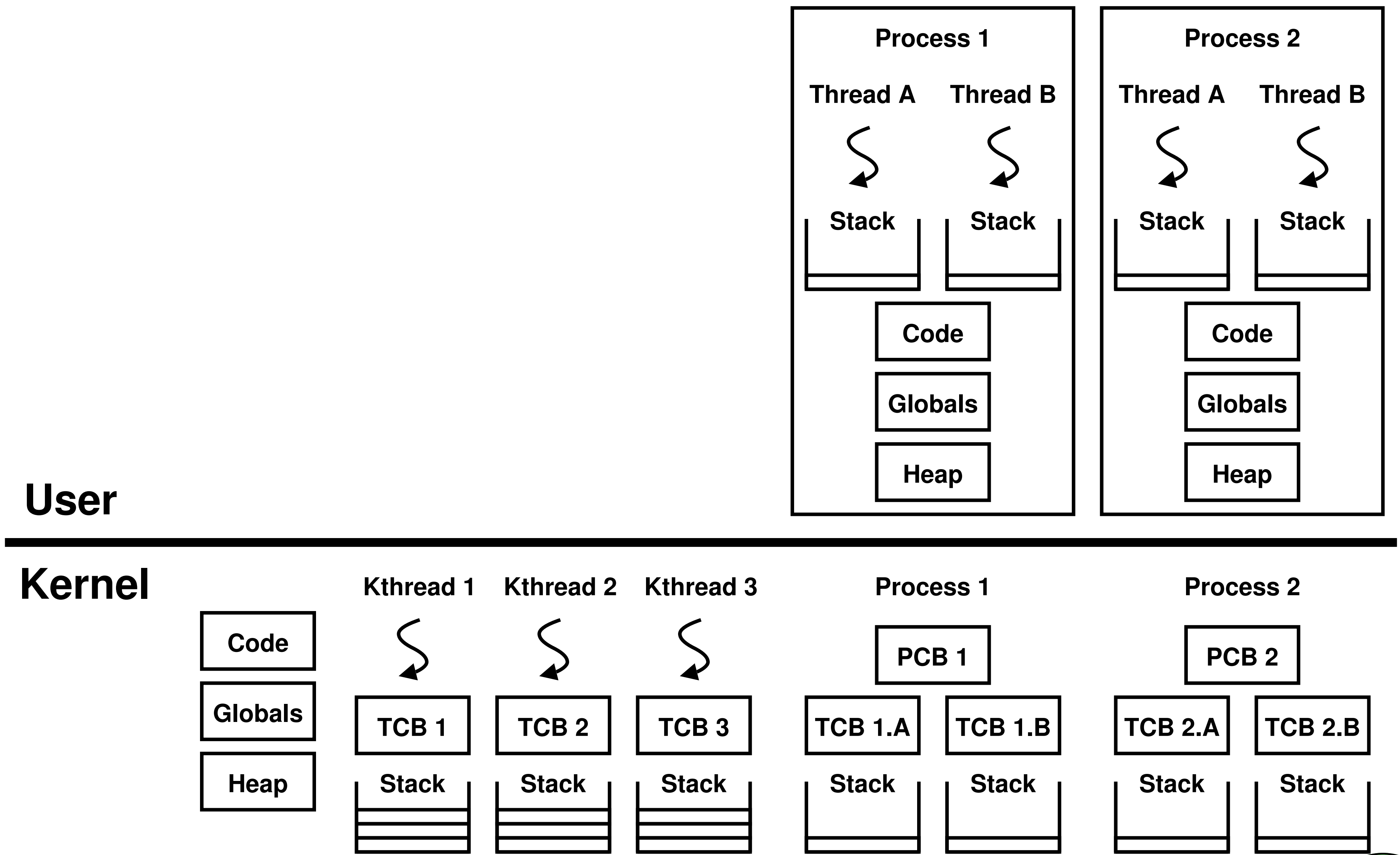
- **Kernel threads**
  - thread abstraction only available to kernel
  - at the user level, we have *single-threaded processes*
    - before user space multithreading was invented, what's running in the user space is just a process (there was no such a thing as a user space thread)
      - the kernel has always been multithreaded
- **Multi-threaded processes using kernel threads (Linux, MacOS)**
  - kernel thread operations available via system calls
- **User-level threads**
  - thread operations without system calls

Kernel Threads and Single-Threaded Processes



- Only PCB in the kernel for a user processe (no TCB)

Multi-threaded Processes Using Kernel Threads



- Each user thread has a corresponding TCB in the kernel

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 if no other thread is available to run)

- `stub(func, args)`
  - call `(*func)(args)`
  - if return, call `thread_exit()`

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

  - `stub()` is also known as the *thread startup function*

Create A Thread

```
void thread_create(thread_t *thread, void (*func)(int), int arg) {
  // allocate TCB and stack
  TCB *tcb = new TCB();
  thread->tcb = tcb;
  tcb->stack_size = INITIAL_STACK_SIZE;
  tcb->stack = new Stack(INITIAL_STACK_SIZE);
  // initialize registers so that when thread is resumed, it will start
  //     running at stub
  // stack starts at the bottom of the allocated region and grows upward
  tcb->sp = tcb->stack + INITIAL_STACK_SIZE;
  tcb->pc = stub;
  // create a stack frame by pushing stub's arguments and start address
  //     onto the stack: func, arg
  *(tcb->sp) = arg;
  tcb->sp--;
  *(tcb->sp) = func;
  tcb->sp--;
  // create another stack frame so that thread_switch() works correctly
  // this routine is explained after we discuss thread_switch()
  thread_dummySwitchFrame(tcb);
  tcb->state = READY;
  readyList.add(tcb); // put tcb on ready list
}
```

Deleting A Thread

- Two steps to deleting a thread when a thread calls `thread_exit()`
  - remove the thread from the ready list so that it will never run again
  - free the per-thread state allocated for the thread
    - cannot do this immediately, what if you get an interrupt, how can you resume this thread
    - also, the thread cannot finish running the code in `thread_exit()` if it doesn't have a stack?
    - solution: let another thread free the state of this thread when this thread is no longer running
      - can do this in when this other thread is about to return from `thread_join()`
      - last thing in `thread_exit()` is to indicate that this thread is dead and yield the processor

Thread Context Switch

- Voluntary
  - `thread_yield()`
  - `thread_join()` (if child is not done yet)
- Involuntary
  - interrupt or exception
  - some other thread with higher priority wants to run

Voluntary Kernel 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
  - thread switch is always between kernel threads, not between user process and kernel thread (i.e., no thread switching when a system call is made)

```
// we enter as oldThread, but we return as newThread
// returns with new's registers and stack
void thread_switch(oldThreadTCB, newThreadTCB) {
  pushad; // push general register values onto the old stack
  oldThreadTCB->sp = %esp; // save the old thread¢s stack pointer
  runningThread = newThreadTCB; // runningThread is a global variable
  %esp = newThreadTCB->sp; // switch to the new stack
  popad;  // pop register values from the new stack
}
```

Voluntary Kernel Thread Context Switch

```
void thread_yield() {
  TCB *chosenTCB, *finishedTCB;
  // prevent an interrupt from stopping us in the middle of a switch
  disableInterrupts();
  // choose another TCB from the ready list
  chosenTCB = readyList.getNextThread();
  if (chosenTCB == NULL) {
    // nothing else to run, so go back to running the original thread
  } else {
    // move running thread onto the ready list
    runningThread->state = ready;
    readyList.add(runningThread);
    thread_switch(runningThread, chosenTCB);
    // switch to the new thread
    runningThread->state = running;
  }
  // delete any threads on the finished list
  while ((finishedTCB = finishedList->getNextThread()) != NULL) {
    delete finishedTCB->stack;
    delete finishedTCB;
  }
  enableInterrupts();
}
```

Voluntary Kernel Thread Context Switch

- All threads must call `thread_switch()` to give up the processor in the same way as in `thread_yield()`
  - `thread_join()` must call `thread_switch()` with interrupt disabled before it blocks
  - when a thread wants to wait for I/O device to complete, it must add itself to the I/O queue and call `thread_switch()` with interrupt disabled before it blocks
  - how would you switch to a newly created thread (which has never called `thread_switch()`)?
    - you need to create a dummy frame to pretend that it has already called `thread_switch()`

```
// when another thread switches to a newly created thread,
//     the last two lines of thread_switch() work correctly
void thread_dummySwitchFrame(newThread) {
  *(tcb->sp) = stub;
  // return to the beginning of stub
  tcb->sp--;
  tcb->sp -= SizeOfPopad;
}
```

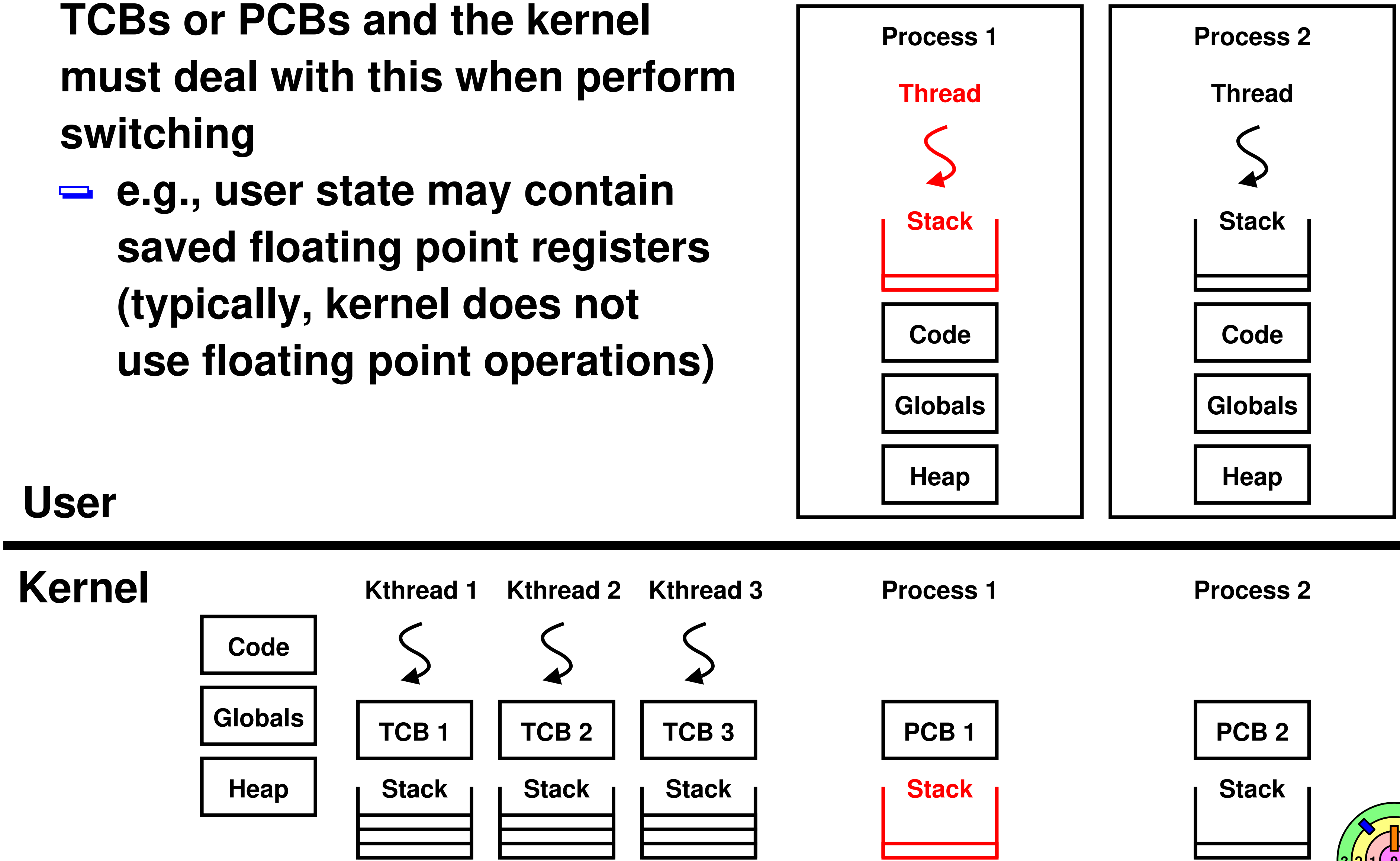
Involuntary Kernel Thread Context Switch

- Timer or I/O interrupt
  - in Ch 2, we have talked about what happens when an interrupt interrupts a running user-level process (involves mode switching and `iret` to go back to user space)
  - the mechanism is almost identical when an interrupt triggers a kernel thread switch (except no mode switching and `iret` is replaced by a regular return from a function)
    - the current thread state is pushed onto the current stack, starting from the current stack pointer
    - to resume a different thread when returning, just call `thread_switch()` right before the handler returns
    - on x86 processor, the `iret` function will examine the saved state in the stack (EFLAGS) to see if it's returning to user space; if it's not returning to user space, it will perform a regular function return)

(4.7) Combining Kernel Threads and Single-Threaded User Processes

Kernel Threads and Single-Threaded Processes

- The ready list contains a mix of TCBs or PCBs and the kernel must deal with this when perform switching
  - e.g., user state may contain saved floating point registers (typically, kernel does not use floating point operations)

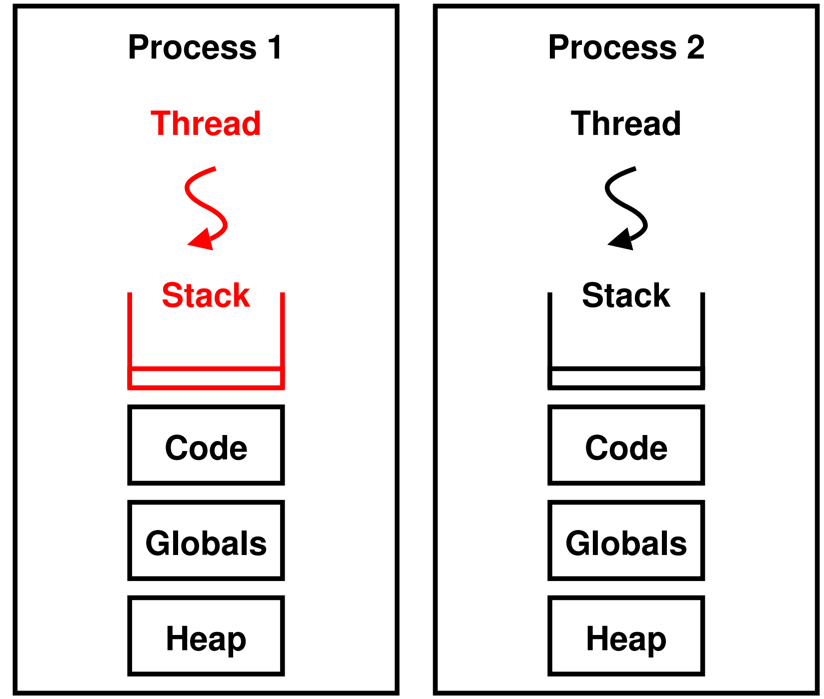


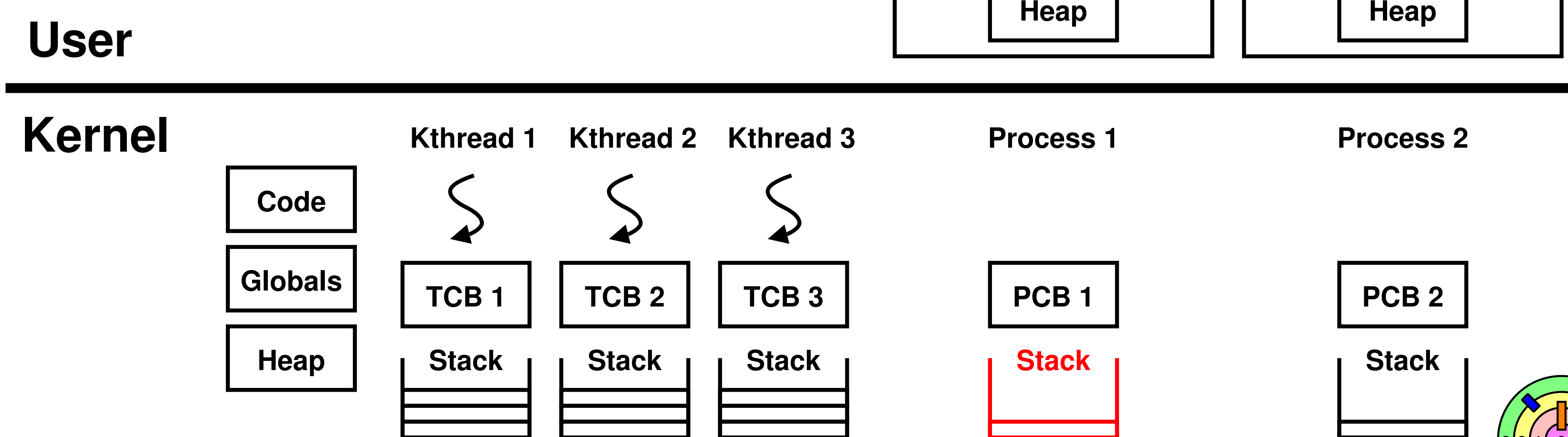
Kernel Threads and Single-Threaded Processes

- The ready list contains a mix of ... el ... erform

> kernel does not need to save floating pointer registers when switching between kernel threads, but have to save them when switch between processes

  - e.g., user state may contain saved floating point registers (typically, kernel does not use floating point operations)

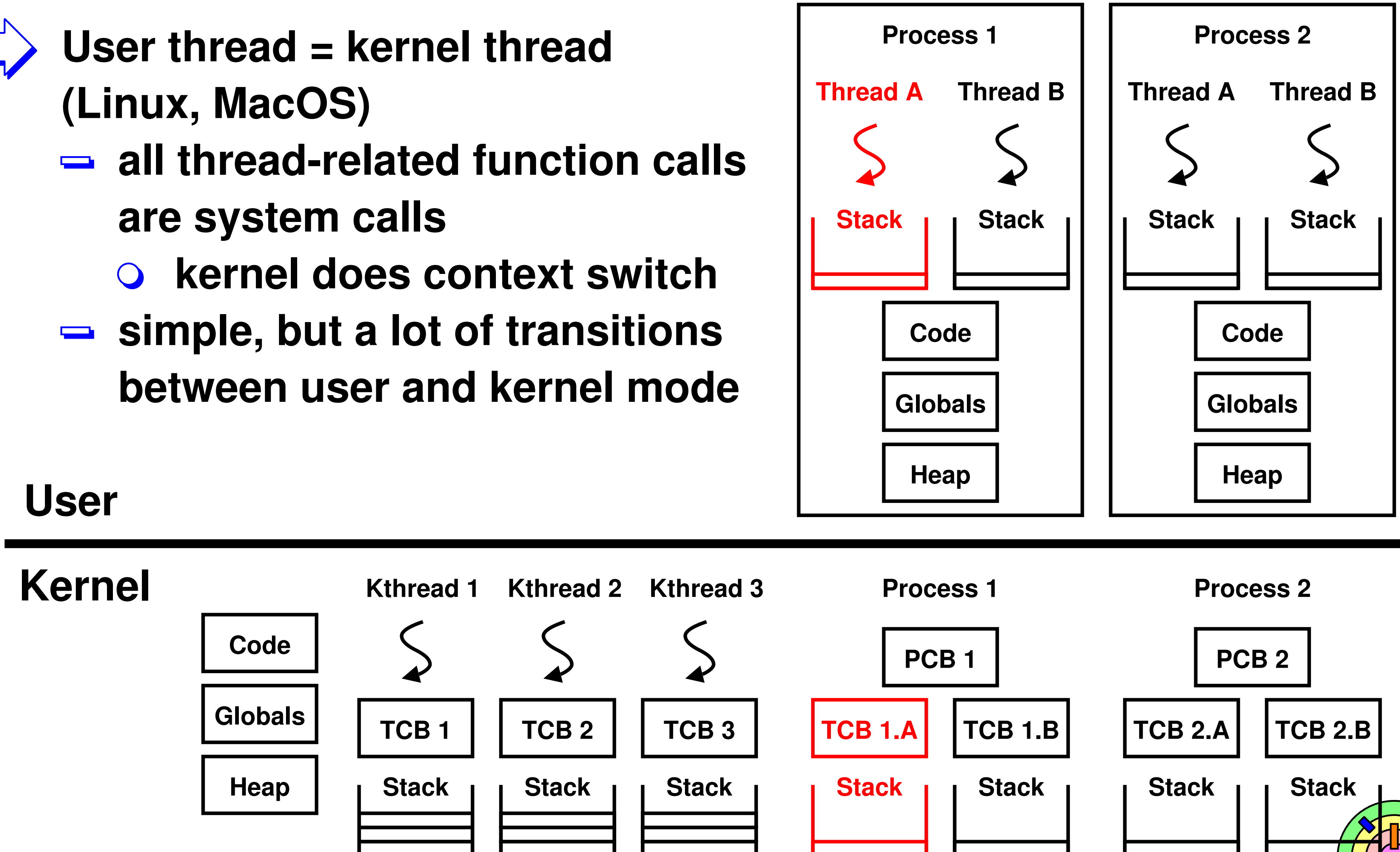




(4.8) Implementing Multi-Threaded Processes

Implementing Multi-threaded User Processes Using Kernel Threads

- User thread = kernel thread (Linux, MacOS)
  - all thread-related function calls are system calls
    - kernel does context switch
  - simple, but a lot of transitions between user and kernel mode

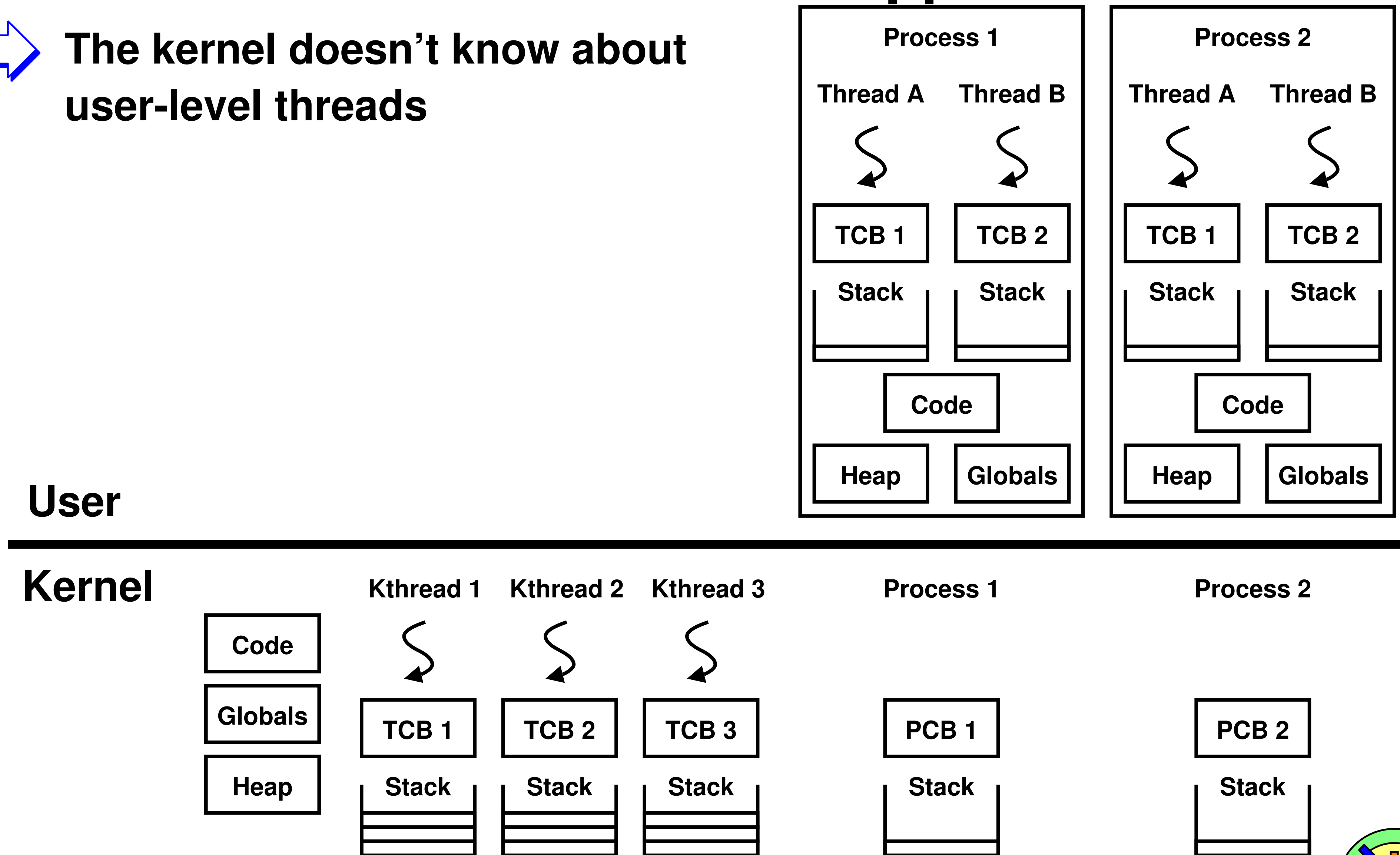


Implementing User-Level Threads Without Kernel Support

- Implement user-level threads completely at user level, without any OS support
  - e.g., *green threads* in the earliest implementation of Sun’s Java Virtual Machine (JVM)
    - to the kernel, a multi-threaded application using green threads appears to be a normal single-threaded process
    - if a user thread makes a system call and get blocked waiting for I/O, the kernel cannot run a different user thread
      - to get true parallelism, you have to run multiple processes

Implementing User-Level Threads Without Kernel Support

- The kernel doesn't know about user-level threads



Implementing User-Level Threads Without Kernel Support

- **Preemptive user-level threads: implementation for process P**
  - user-level thread library makes a system call to register a timer signal handler and signal stack with the kernel
  - when a hardware timer interrupt occurs, the hardware saves P's register state and runs the kernel's handler
  - instead of restoring P's register state and resuming P where it was interrupted, the kernel's handler copies P's saved registers onto P's signal stack
  - the kernel resumes execution in P at the registered signal handler on the signal stack
  - the signal handler copies the processor state of the preempted user-level thread from the signal stack to that thread's TCB
  - the signal handler chooses the next thread to run, re-enables the signal handler (similar to re-enabling interrupts), and restores the new thread's state from its TCB into the processor; execution with the state (newly) stored on the signal stack

Implementing User-Level Threads With Kernel Support

- Today, most programs use kernel-supported threads rather than pure user-level threads
  - major operating systems support threads using standard abstractions, so the issue of portability is less of an issue than it once was

Implementing User-Level Threads With Kernel Support

- Various systems take more of a hybrid model (best of both worlds)
  - *hybrid thread join*
  - *per-processor kernel threads*
  - *scheduler activations* (in Windows): user-level thread scheduler is notified/activated for every kernel event that might affect the user-level thread system

(4.9) Alternative Abstractions

- Asynchronous I/O and event-driven programming
  - allows a single-threaded program to cope with high-latency I/O devices by overlapping I/O with processing and other I/O
- Data parallel programming
  - all processors perform the same instruction in parallel on different parts of a data set

Extra Slides