Introduction to Operating Systems

CPSC/ECE 3220 Summer 2018

Lecture Notes
OSPP Chapter 4

(adapted by Mark Smotherman from Tom Anderson's slides on OSPP web site)

Motivation for Threads

- 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

Examples of 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

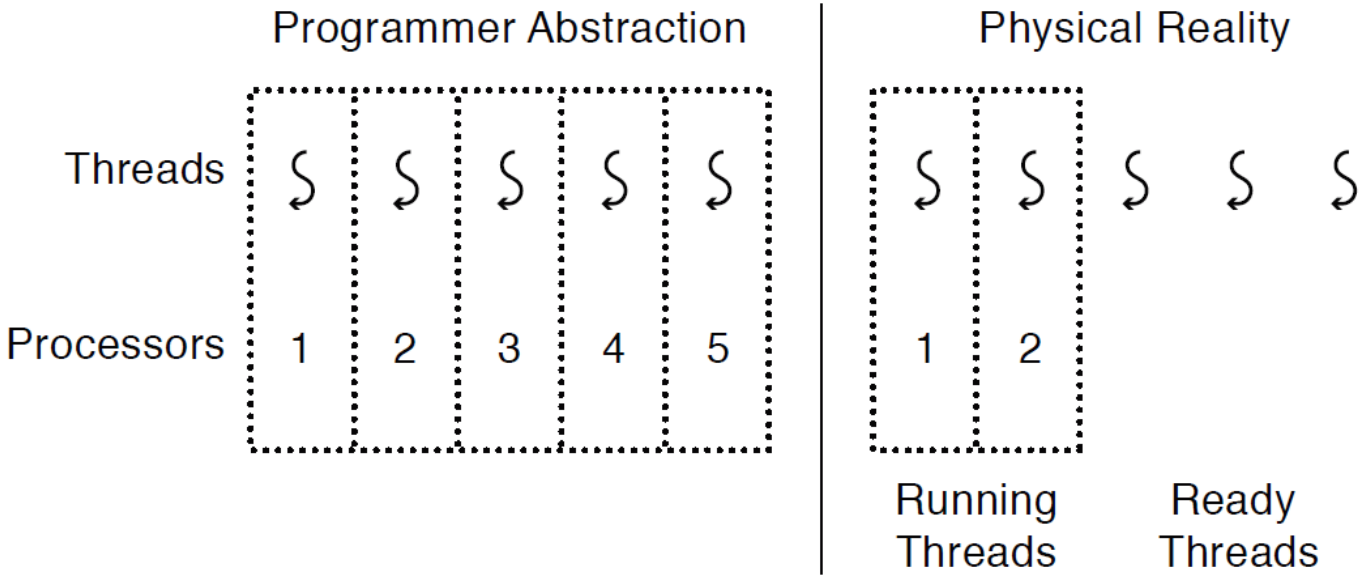
- A thread is a single execution sequence that represents a separately schedulable task
  - Single execution sequence: familiar programming model
  - Separately schedulable: OS can run or suspend a thread at any time
- Protection is an orthogonal concept
  - Can have one or many threads per protection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

Threads vs. Processes

- For parallel processing involving shared objects, threads are more efficient than processes
  - Cheaper to create and destroy
  - Faster to switch among
  - Communicate through shared memory in single process
- But code is hard to get correct when multiple threads can update shared objects
  - “[N]on-trivial multi-threaded programs are incomprehensible to humans.” Edward Lee, UCB
  - “For most purposes proposed for threads, events are better. Threads should be used only when true CPU concurrency is needed.” John Ousterhout

Thread Abstraction

- Infinite number of processors
- Threads execute with variable speed
  - Programs must be designed to work with any schedule
  - Otherwise false assumptions can lead to incorrect results



Programmer vs. Processor View

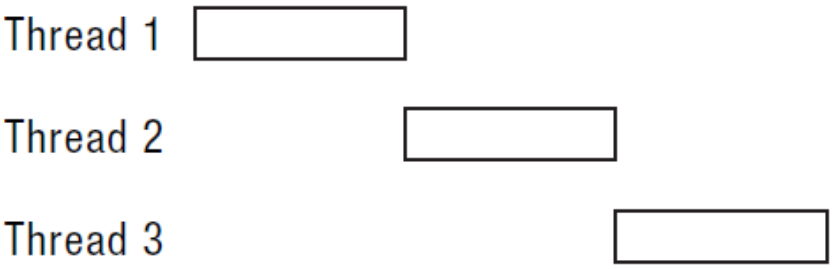
Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
. . .	. . .	. . .	. . .

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

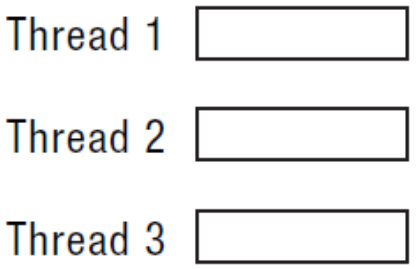
. . .	. . .		

Possible Executions

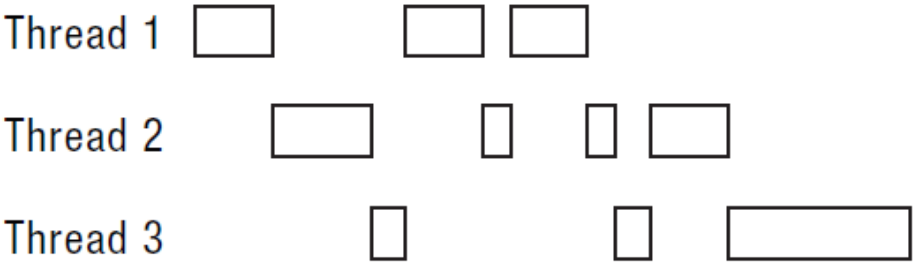
One Execution



Another Execution



Another Execution



Thread Operations

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

Thread Context Switch

- Voluntary
  - Thread_yield()
  - Thread_join() – if child is not done yet
- Involuntary (“preemption”)
  - Interrupt or exception
  - Some other thread is higher priority

Fork/Join Executes a Procedure Call in Parallel

Procedure Call/Return

call func() ✉ func()

execute

 return

execute

- Single path of execution
- Caller resumes execution on return

Thread Fork/Join

t_create() ✉ func()

execute execute

t_join() t_exit()

execute

- Parallel execution
- Exit is immediate
- Join will wait for exit if necessary

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

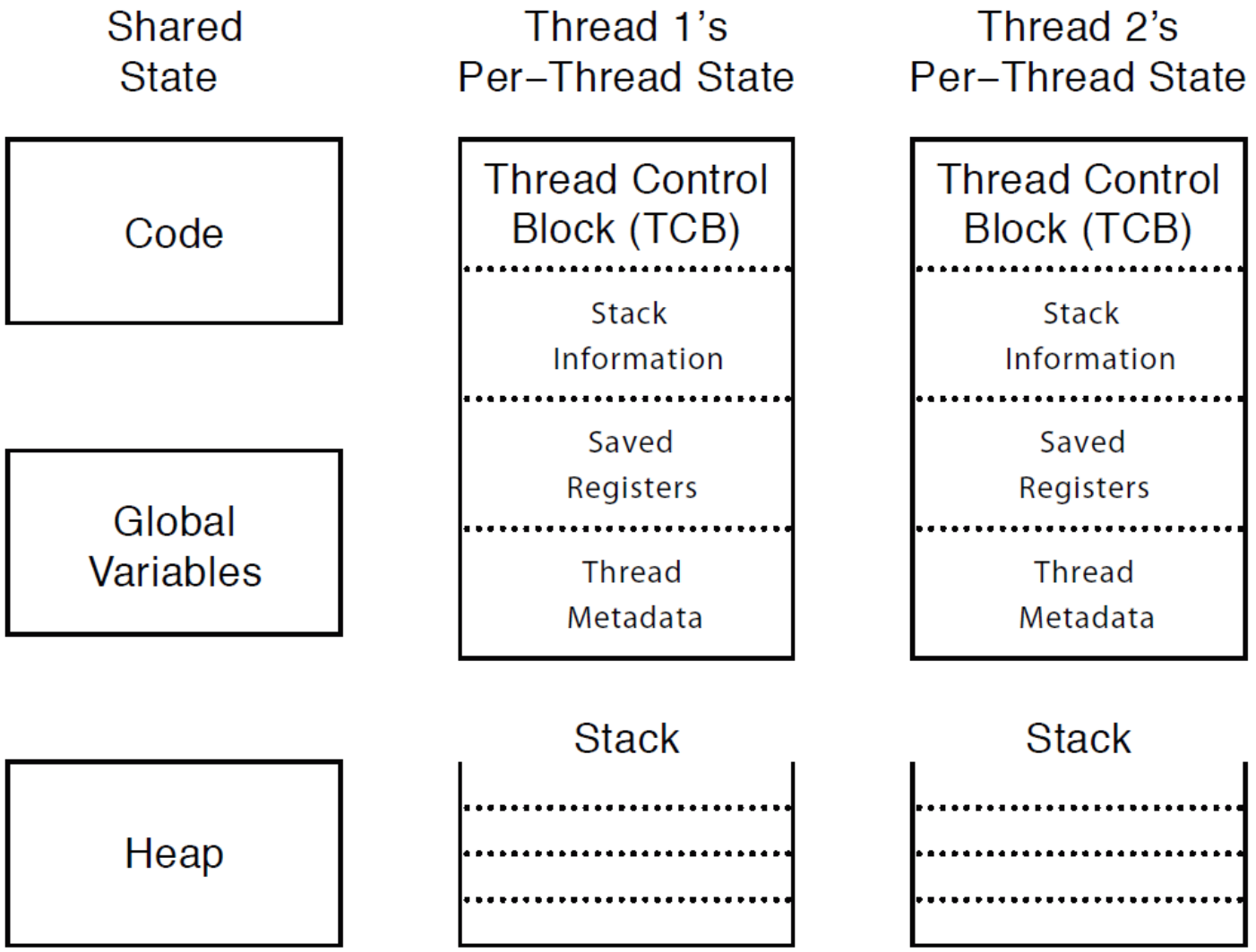
- Threads can create children, and wait for their completion
- Data can be shared before fork and after join
  - Otherwise extra code to coordinate access (Chapter 5)
- Examples:
  - Web server: fork new thread for each new connection
    - As long as the threads are completely independent
  - Merge 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

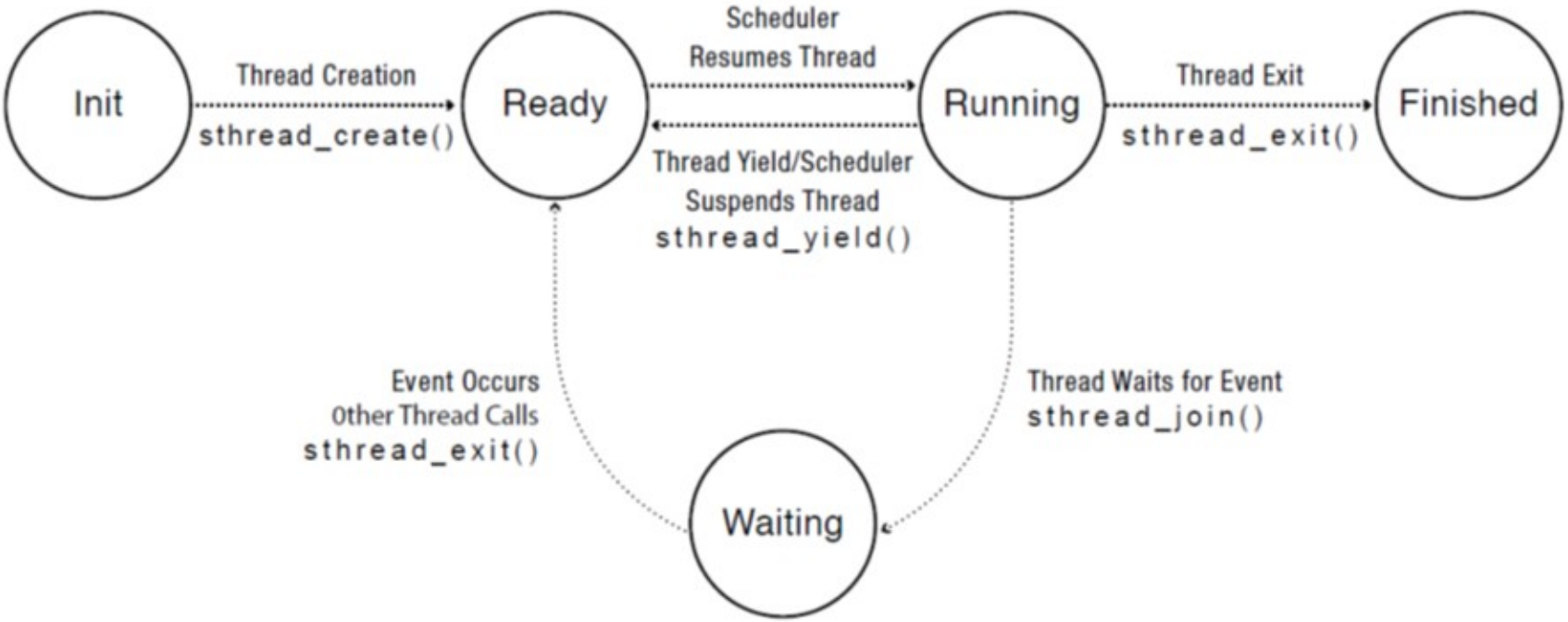


Thread Metadata

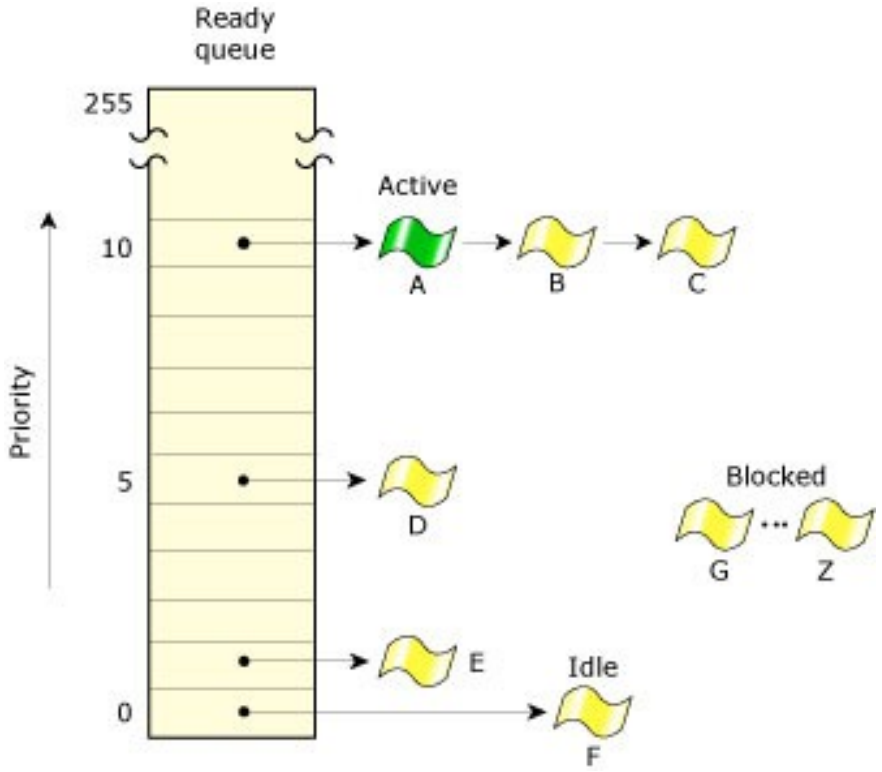
- Thread ID
- Scheduling priority
- Thread state (ready, running, waiting, …)

State of Thread	Location of TCB	Location of Registers
Init	being created	TCB or thread's stack
Ready	Ready List	TCB/stack
Running	Running List	processor
Waiting	synch. variable's Waiting List	TCB/stack
Finished	Finished List then deleted	TCB/stack or deleted

Thread Lifecycle



Ready Threads in QNX



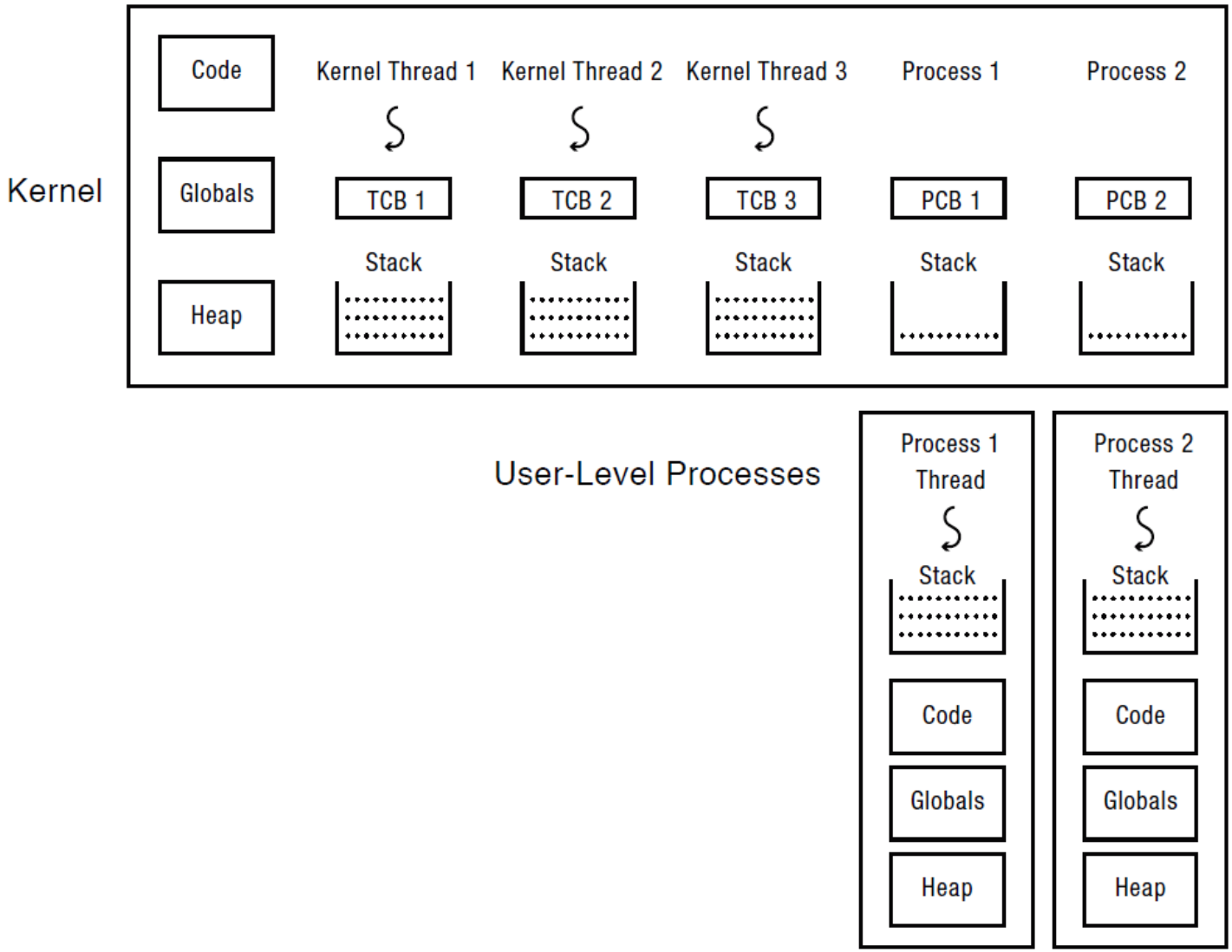
Shows alternate design decisions with one list per priority level and with running thread (“active”) remaining linked

(Diagram from http://www.qnx.com/developers/docs/6.3.2/neutrino/sys_arch/kernel.html)

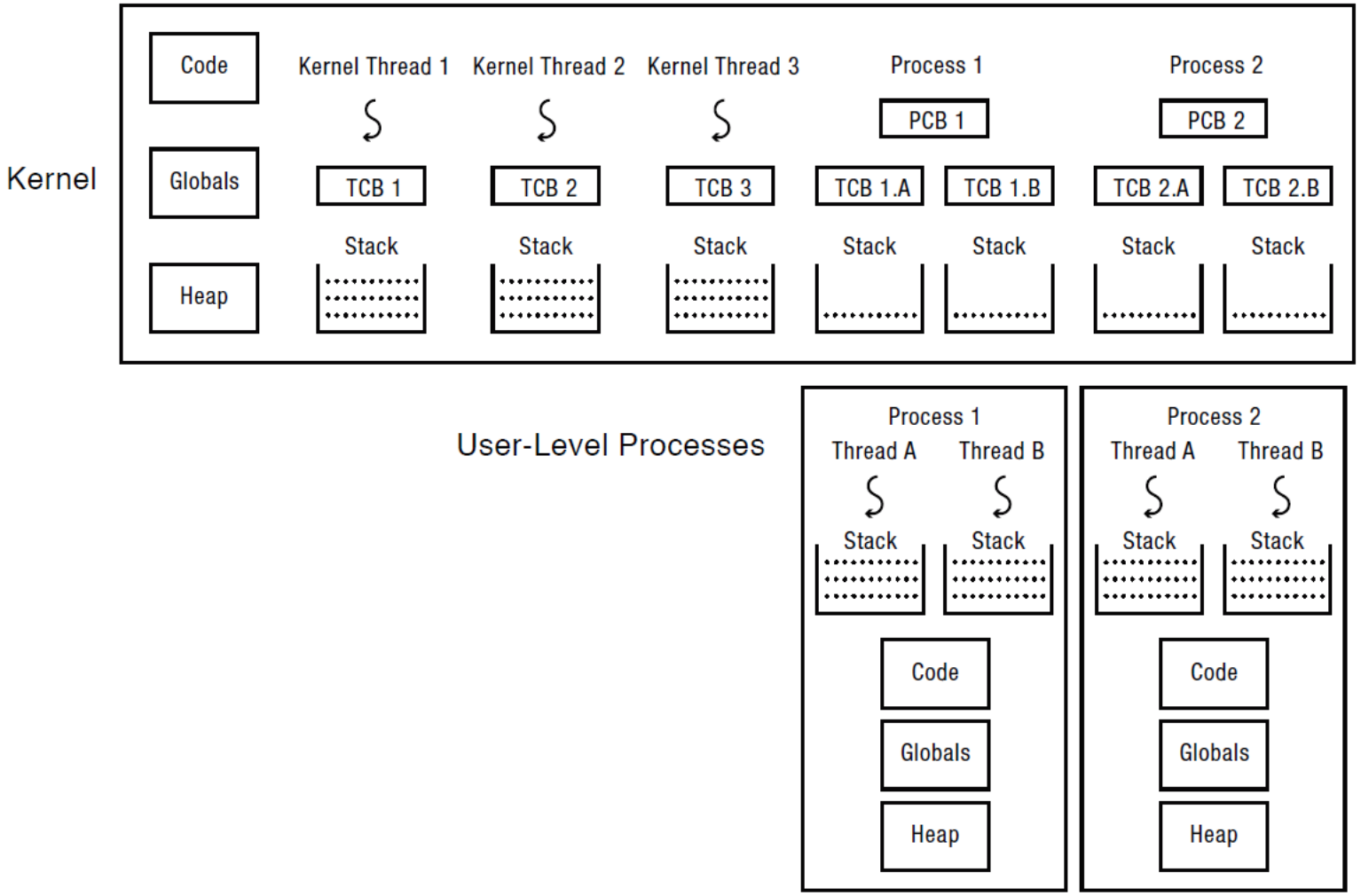
Threads at User and Kernel Level

- 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
- Kernel design
  - Kernel can have zero threads (e.g., IBM MVT)
  - Kernel can use internal threads
  - In general, interrupt handlers are not threads but can be used to wake or signal threads

Multithreaded OS Kernel



Multithreaded User Processes (Take 1)



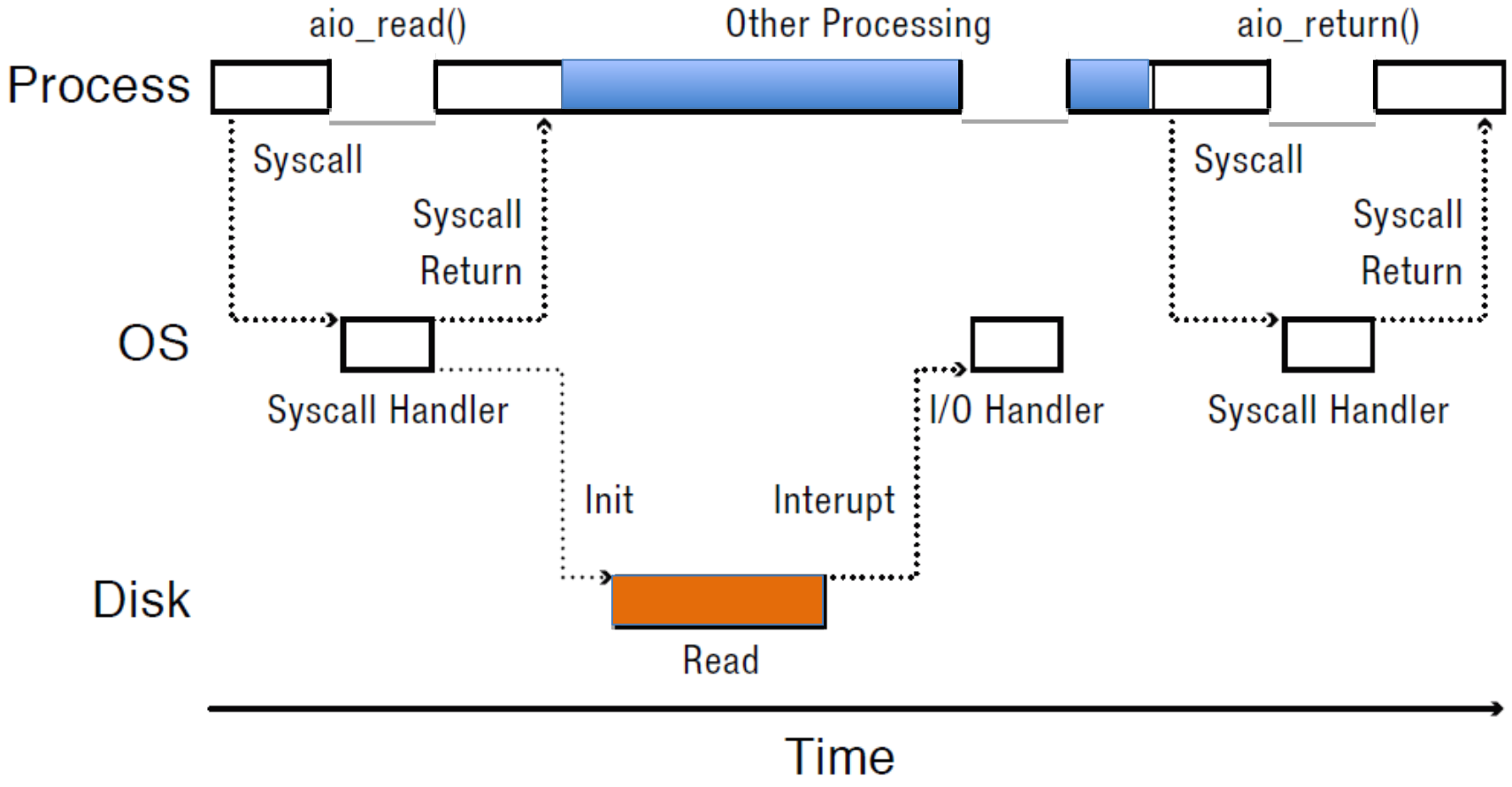
Multithreaded User Processes (Take 2)

- 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 real concurrency
    - 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 needs a user-level scheduling decision
  - Process assigned a new processor
  - Processor removed from process
  - System call blocks in kernel

Asynchronous I/O as a Alternative to Threads



(if time permits)

Event-Driven Approach

- Ousterhout description:
  - One execution sequence; no concurrency
  - Establish callbacks for events
  - Event loop waits for an event and invokes handlers
  - No preemption of event handlers
  - Event handlers are generally short-lived

- But must add “continuation” data structures if event processing is complex and needs local state and tracking of next step

Question

- When is event-driven programming better than multithreaded conc