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

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
- 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

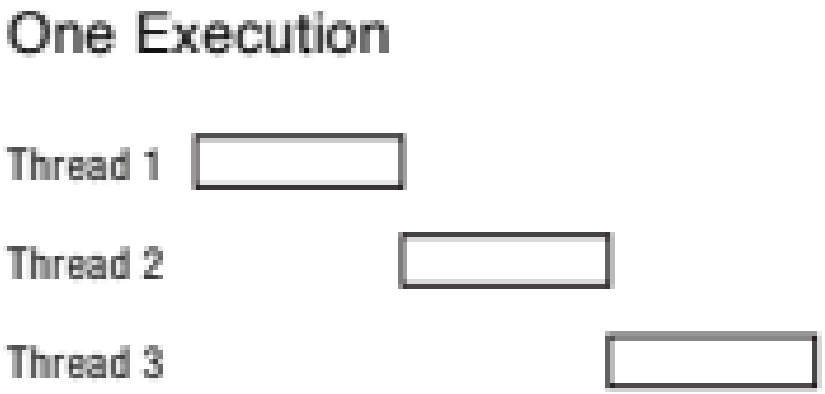
Thread Abstraction

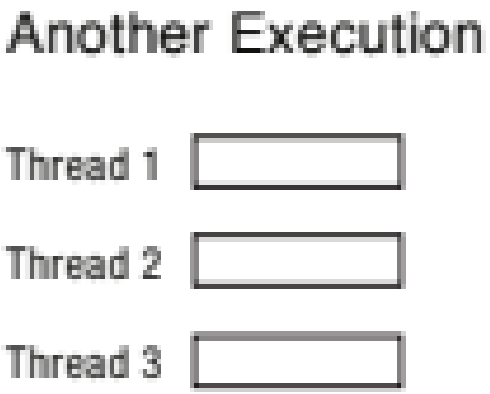
- Infinite number of processors
- Threads execute with variable speed
  - Programs must be designed to work with any schedule

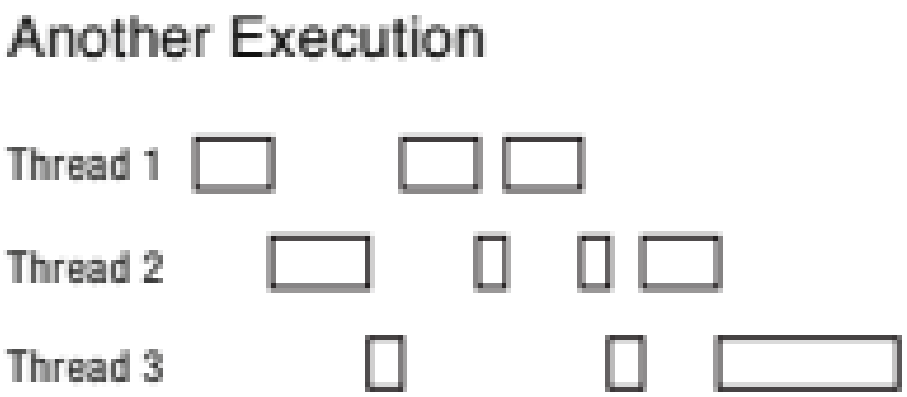
Programmer vs. Processor View

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;`

Possible Executions







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
  - Quit thread and clean up, wake up joiner if any

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 Parallelism

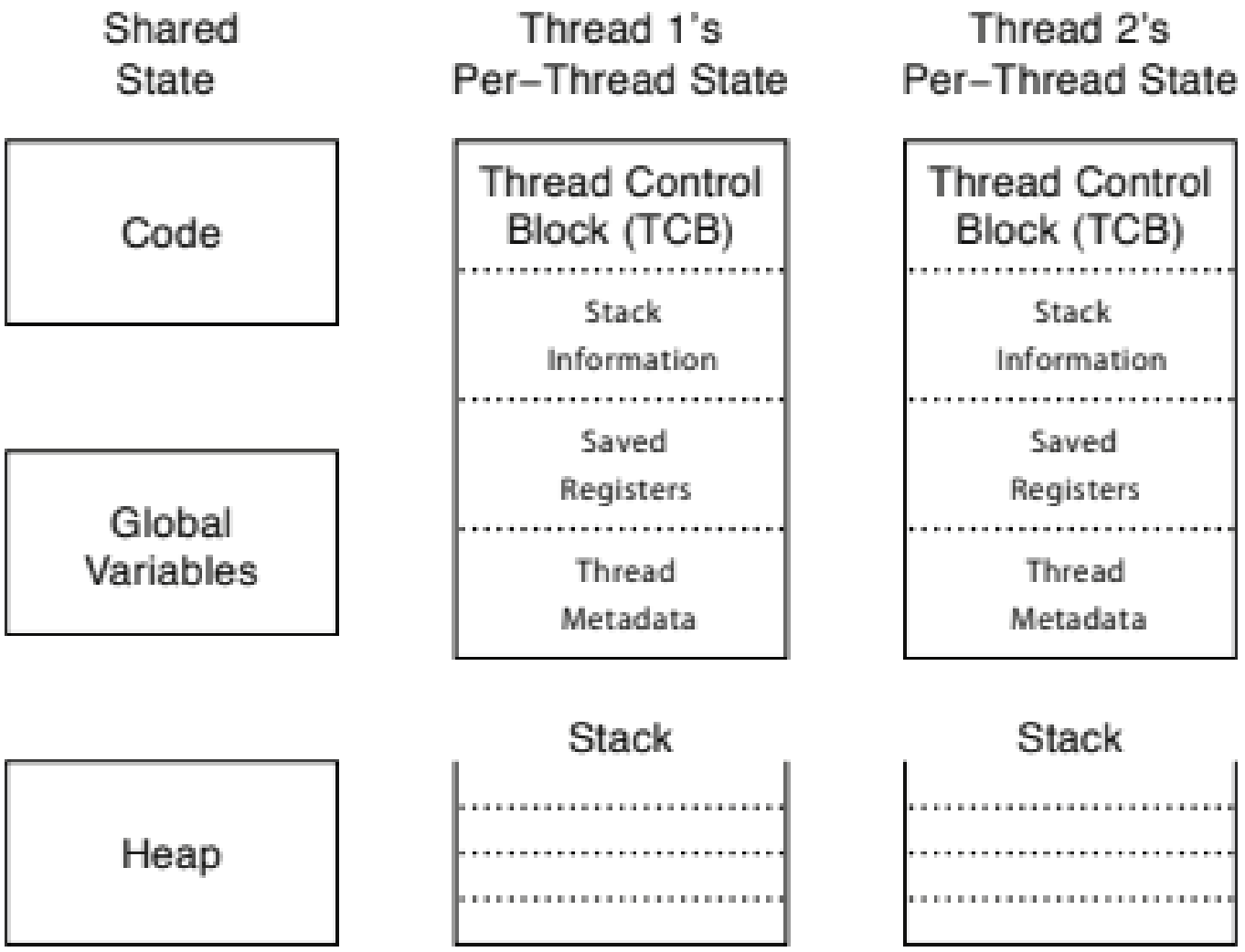
- Threads can create children, and wait for their completion
- Data only shared before fork/after join
- Examples:
  - Web server: fork a new thread for every new connection
    - As long as the threads are completely independent
  - Merge sort
  - Parallel memory copy

bzero with fork/join concurrency

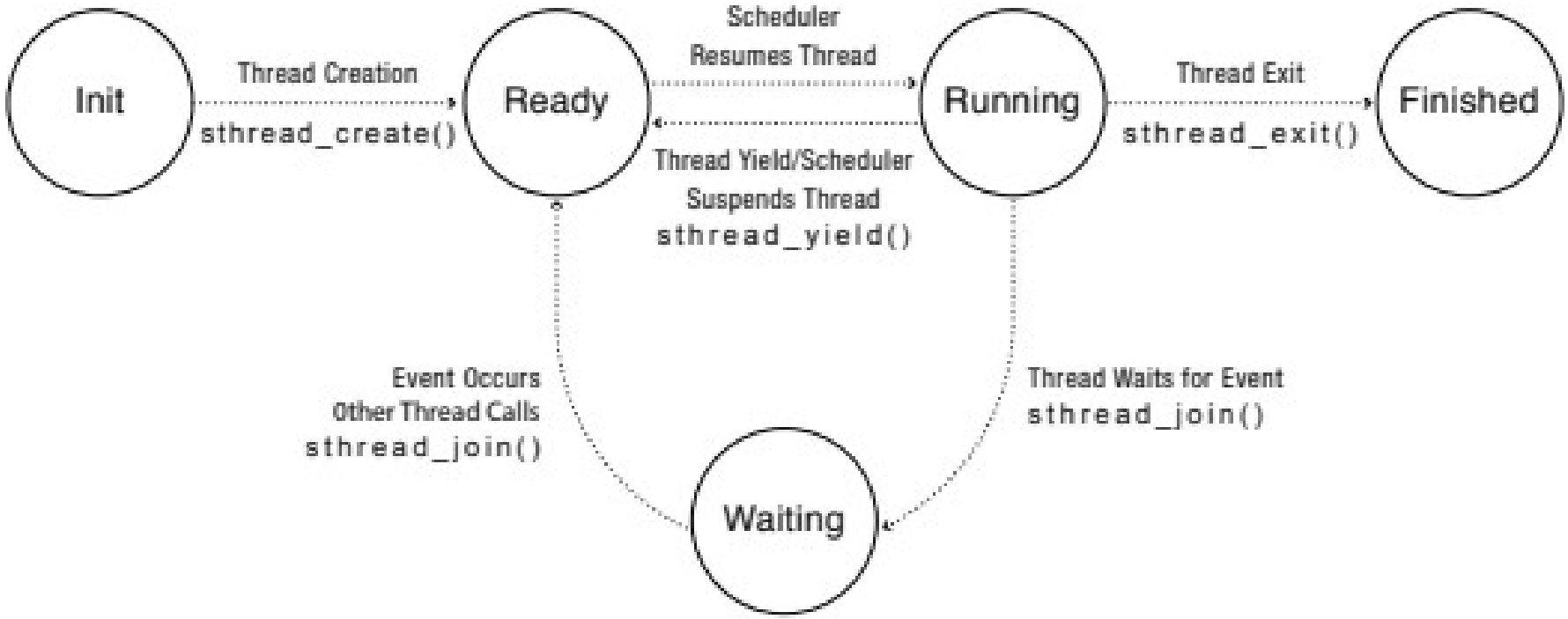
```
void blockzero (unsigned char *p, int length) {
    int i, j;
    thread_t threads[NTHREADS];
    struct bzeroparams params[NTHREADS];

// 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



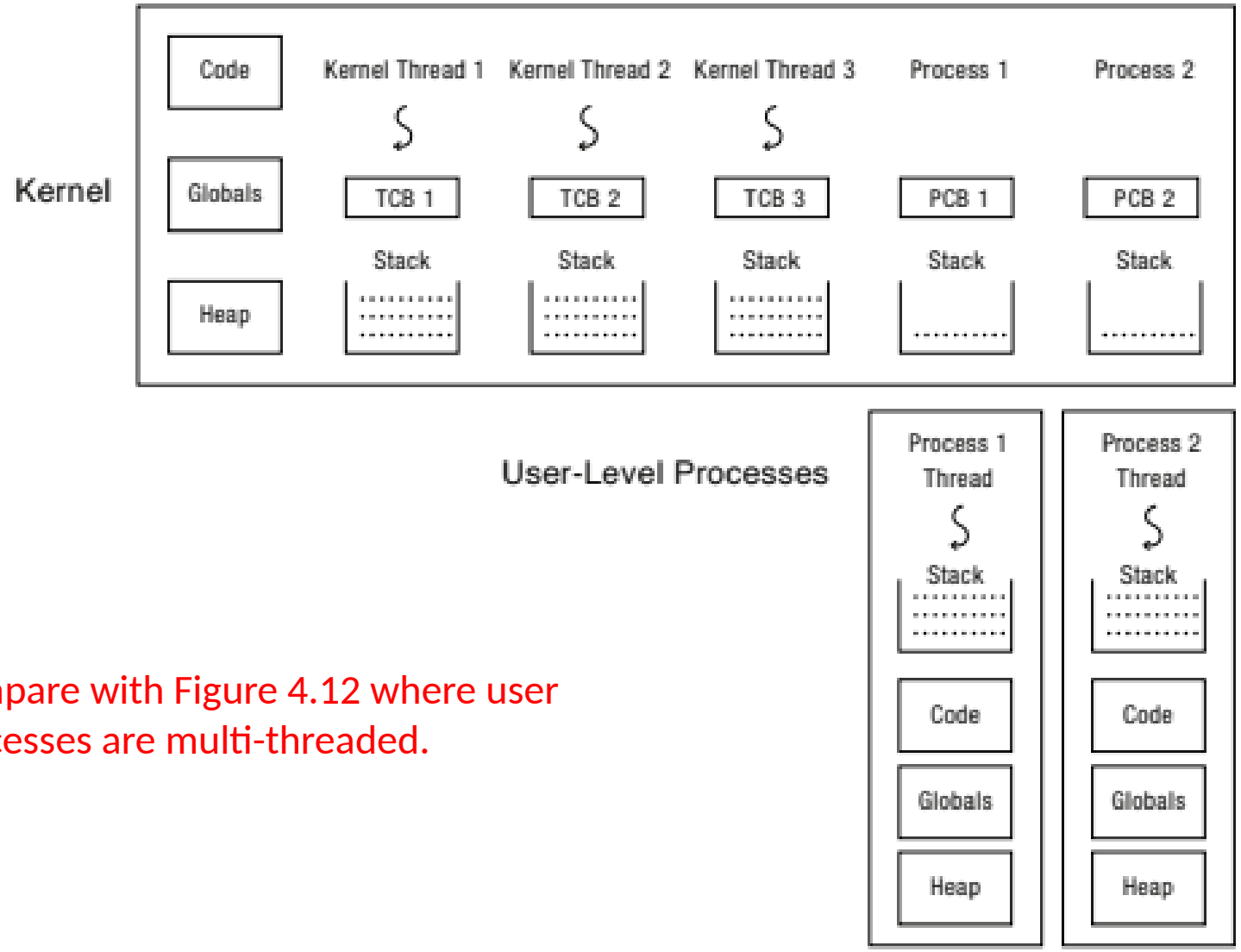
Thread Lifecycle



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



Compare with Figure 4.12 where user processes are multi-threaded.

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):
  - Call (*func)(args)
  - If return, call thread_exit()

See Figure 4.13 for details

Thread Removal

- Can a thread remove itself from the ReadyList?
  - Issue?
  - Solution?

Thread Context Switch

- Voluntary
  - Thread_yield
  - Thread_join (if child is not done yet)
  - Thread_exit
- Involuntary
  - Timer interrupt or exception
  - I/O hardware events
  - 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

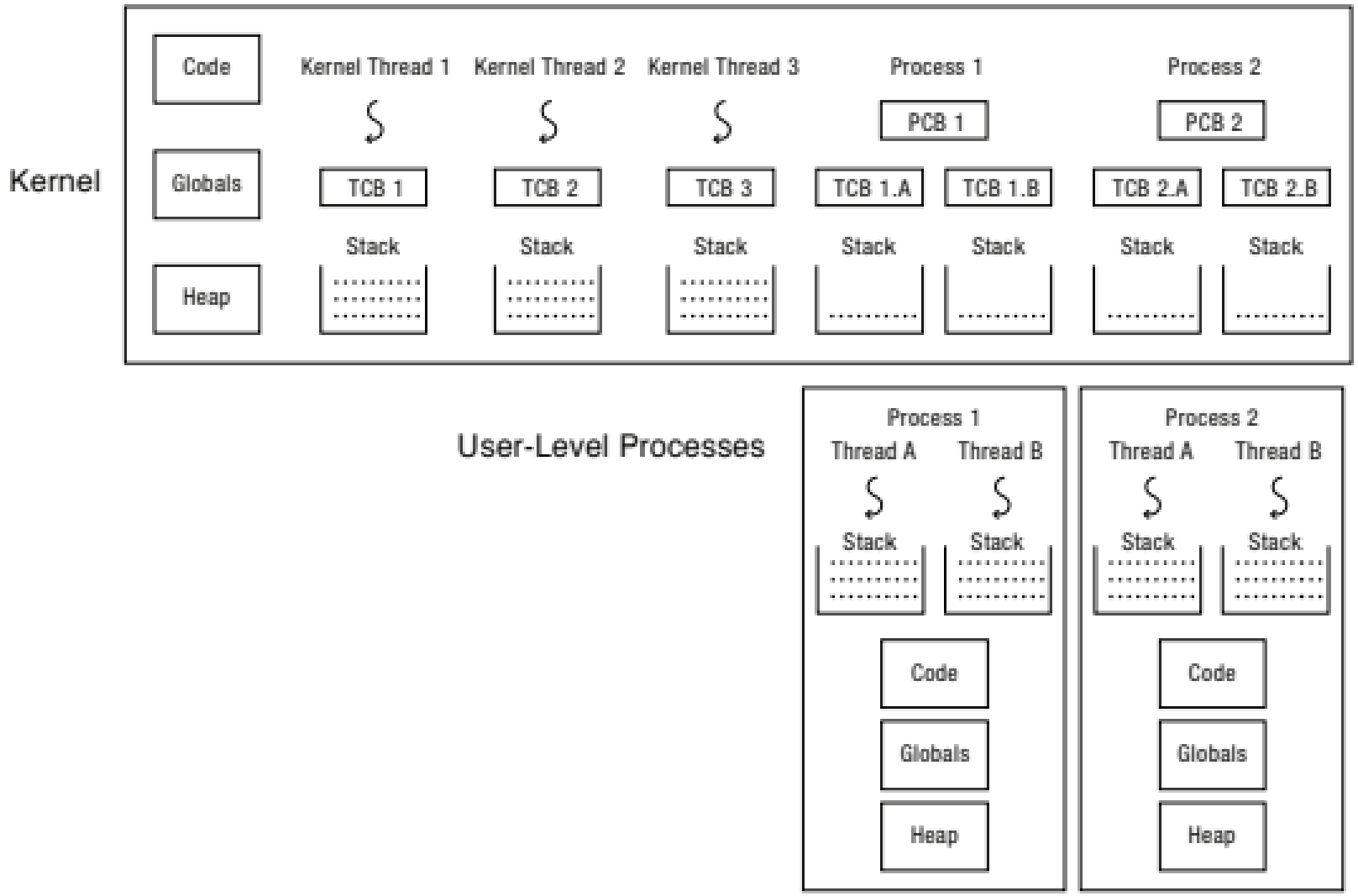
Two Threads Call Yield

Thread 1's instructions	Thread 2's instructions	Processor's instructions
"return" from thread_switch into stub		"return" from thread_switch into stub
call go		call go
call thread_yield		call thread_yield
choose another thread		choose another thread
call thread_switch		call thread_switch
save thread 1 state to TCB		save thread 1 state to TCB
load thread 2 state		load thread 2 state
	"return" from thread_switch into stub	"return" from thread_switch into stub
	call go	call go
	call thread_yield	call thread_yield
	choose another thread	choose another thread
	call thread_switch	call thread_switch
	save thread 2 state to TCB	save thread 2 state to TCB
	load thread 1 state	load thread 1 state
return from thread_switch		return from thread_switch
return from thread_yield		return from thread_yield
call thread_yield		call thread_yield
choose another thread		choose another thread
call thread_switch		call thread_switch

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
  - Better portability
  - Library does thread context switch
  - Limitation
  - Preemption via upcall/UNIX signal on timer interrupt

User-Level Threads without Kernel Support

- JVM
  - Minimize dependencies on specific Oss
  - Maximize protability
  - Green threads, i.e, pure user-level threads

- Idea: thread library instantiates all of its data structures within the process and then make just procedure calls to the thread library.

- Limitations
  - OS kernel is unaware of user-level ready list state
  - What if the application runs a blocking call for I/O?
  - Can be fixed by preemption among user-level threads

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

Alternative Abstractions

- Asynchronous I/O
  - Issue multiple concurrent I/O requests at the same time
  - I/O system calls returns immediately
  - At a later time, OS provides the results by
    - Calling a signal handler
    - Placing the result in a queue in the process's memory
    - Storing the result in kernel until another system call from the process

Alternative Abstractions

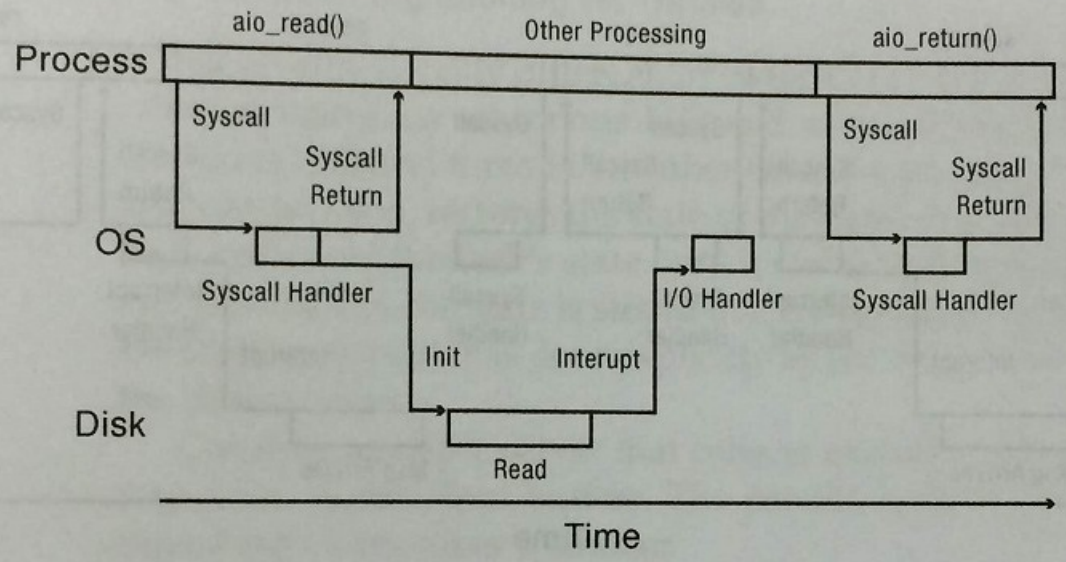


Figure 4.16: An asynchronous file read on Linux. The application calls aio_read to start the read; this system call returns immediately after the disk read is initialized. The application may then do other processing while the disk is completing the requested operation. The disk interrupts the processor when the operation is complete; this causes the kernel disk interrupt handler to run. The application at any time may ask the kernel if the results of the disk read are available, and then retrieve them with aio_return.

Alternative Abstractions

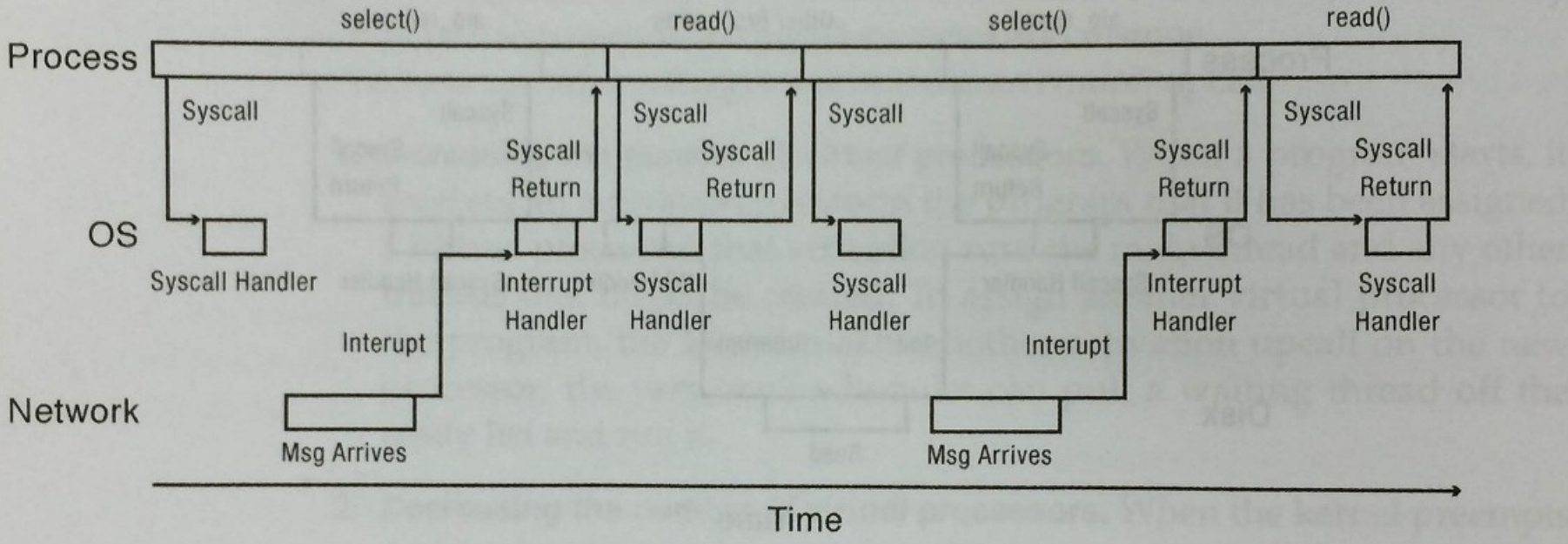


Figure 4.17: A server managing multiple concurrent connections using select. The server calls select to wait for data to arrive on any connection. The server then reads all available data, before returning to select.

Event Driven Programming

- Derived from asynchronous I/O
- One thread with a loop
  - Each iteration gets and processes the next I/O event.
  - Uses a data structure called *continuation*.
- Example: Web Server
  - Makes a network connection
  - Reads a request from network connection
  - Reads the requested data from disk
  - Writes the requested data to the network connection

Event Driven Programming vs Threads

- Coping with high latency devices
- Exploiting multiple processors
- Responsiveness
- Program structure

Thread programming often more natural, reasonably efficient, and simpler when running on multiple processors.