CONCURRENCY

Slides adapted from Tom Anderson's "Operating Systems: Principles and Practice"

Motivation

- Operating systems need to be able to handle multiple things at once
  - processes, interrupts, background system maintenance
- Servers need to handle MTAO
  - Multiple connections handled simultaneously
- Parallel programs need to handle MTAO
  - To achieve better performance
- Programs with user interfaces often need to handle MTAO
  - To achieve user responsiveness while doing computation
- Network and disk bound programs need to handle MTAO
  - To hide network/disk latency

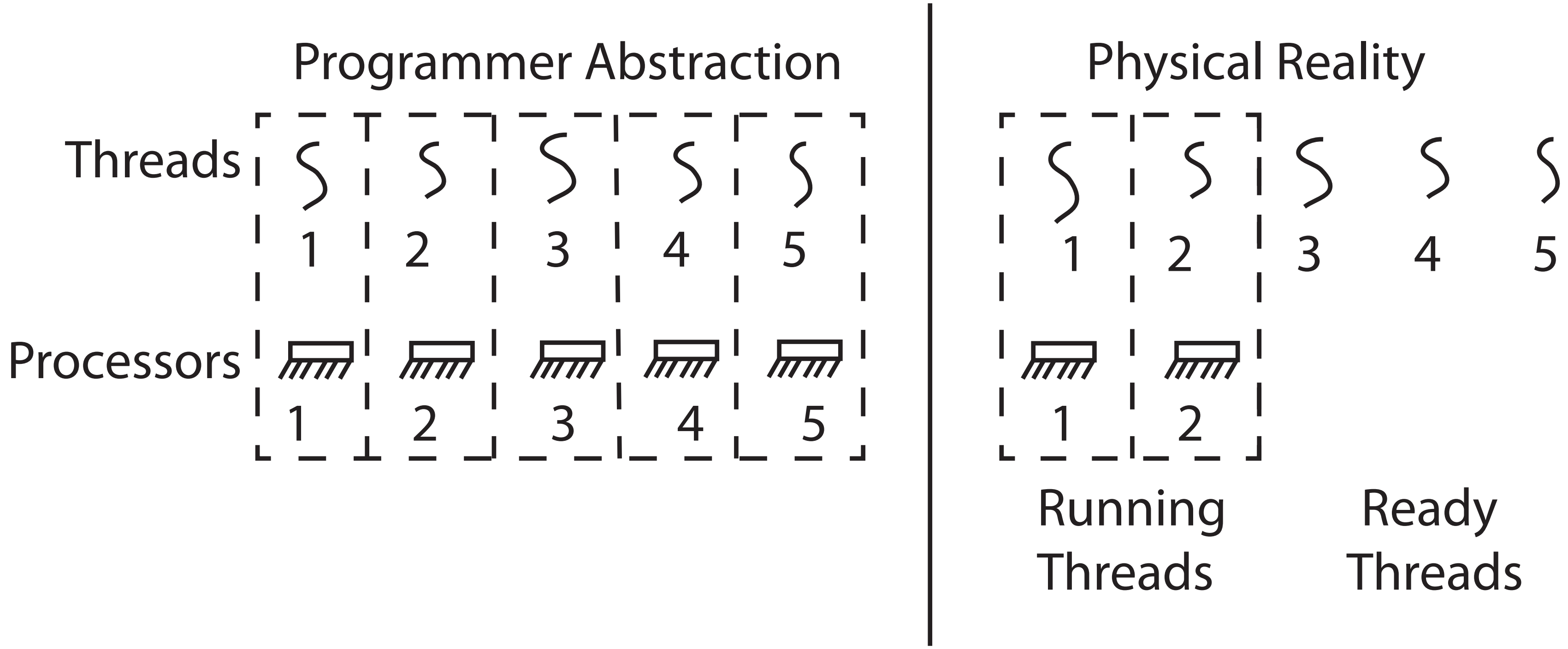
Definitions

- A thread is a single execution sequence that represents a separately schedulable task

- Protection is an orthogonal concept
  - Can have one or many threads per protection domain
    - Single threaded user program: one thread, one protection domain
    - Multi-threaded user program: multiple threads, sharing same data structures, isolated from other user programs
    - Multi-threaded kernel: multiple threads, sharing kernel data structures, capable of using privileged instructions

Thread Abstraction

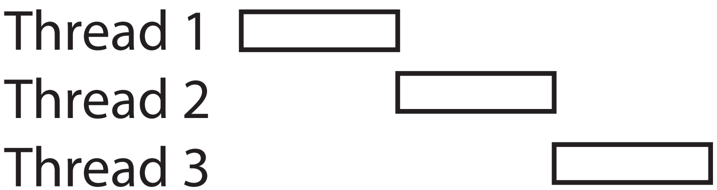
- Infinite number of processors
- Threads execute with variable speed
  - Programs must be designed to work with any schedule



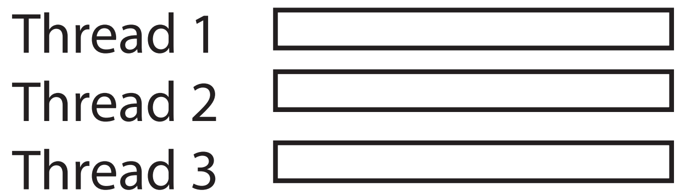
Programmer vs. Processor View

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

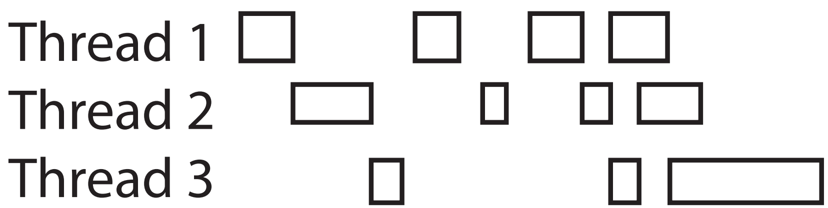
Possible Executions



a) One execution



b) Another execution



c) Another execution

Thread API

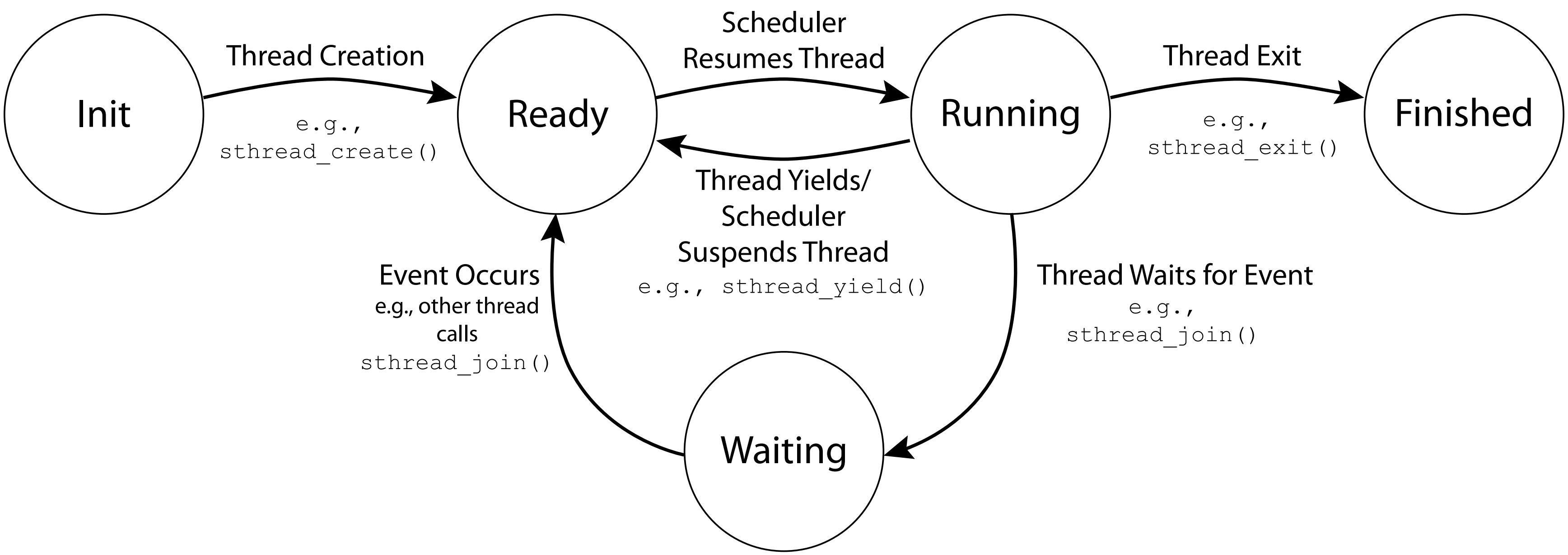
- thread_create(func, args)
  - Create a new thread to run func(args)
- thread_yield()
  - Relinquish processor voluntarily
- thread_join(thread)
  - In parent, wait for forked thread to exit, then return
- thread_exit()
  - Quit thread and clean up, wake up joiner if any

Main: Fork 10 threads call join on them, then exit

- What other interleavings are possible?
- What is maximum # of threads running at same time?
- 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.
```

Thread Lifecycle

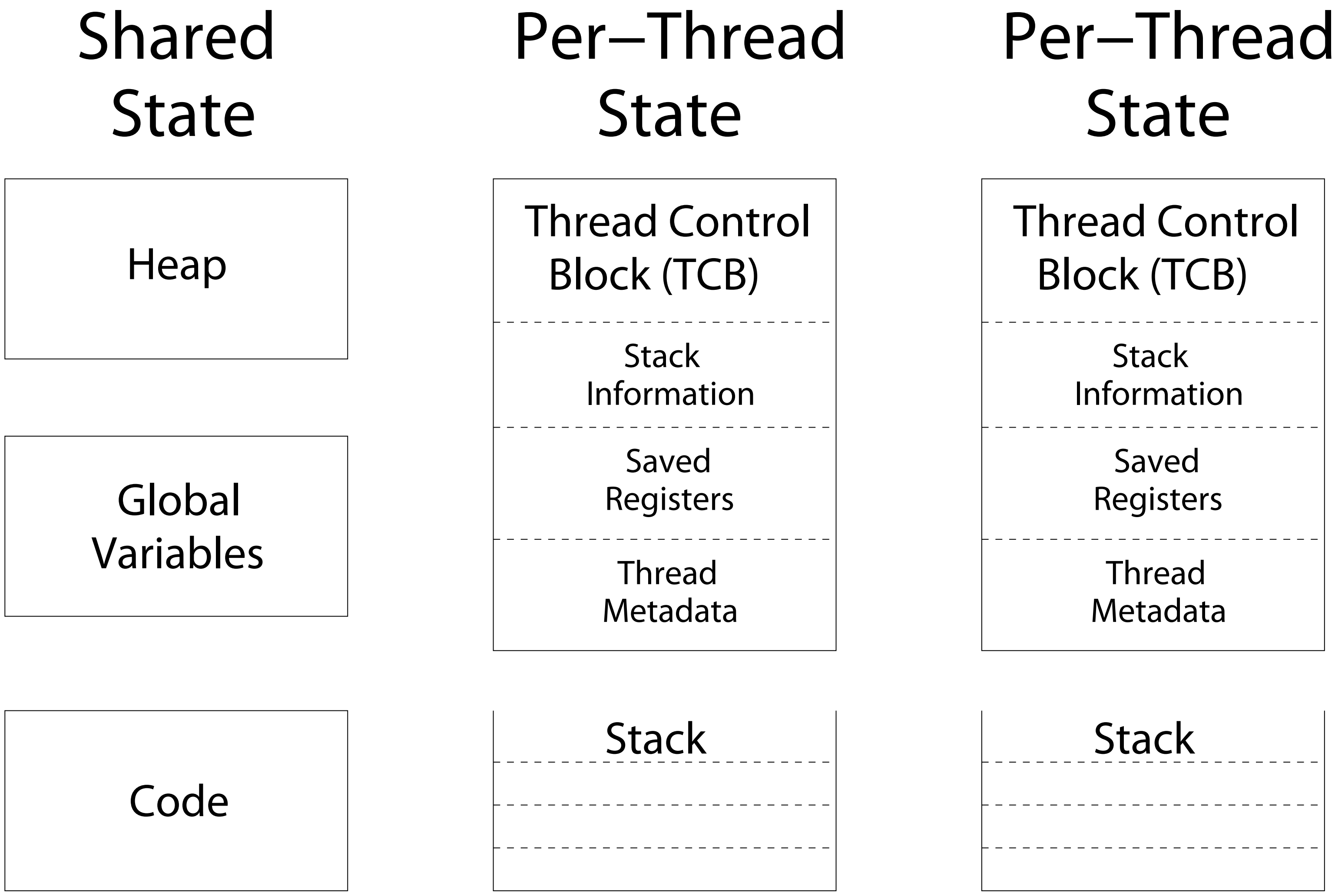


Slides adapted from Tom Anderson's "Operating Systems: Principles and Practice"

Thread Control Block

- Stack
  - What if a thread puts too many procedures on its stack?
    - What should happen?
    - What happens in Java?
    - What happens in Linux?
- Copy of processor registers
- Metadata
  - Id
  - Priority
  - Status
  - …

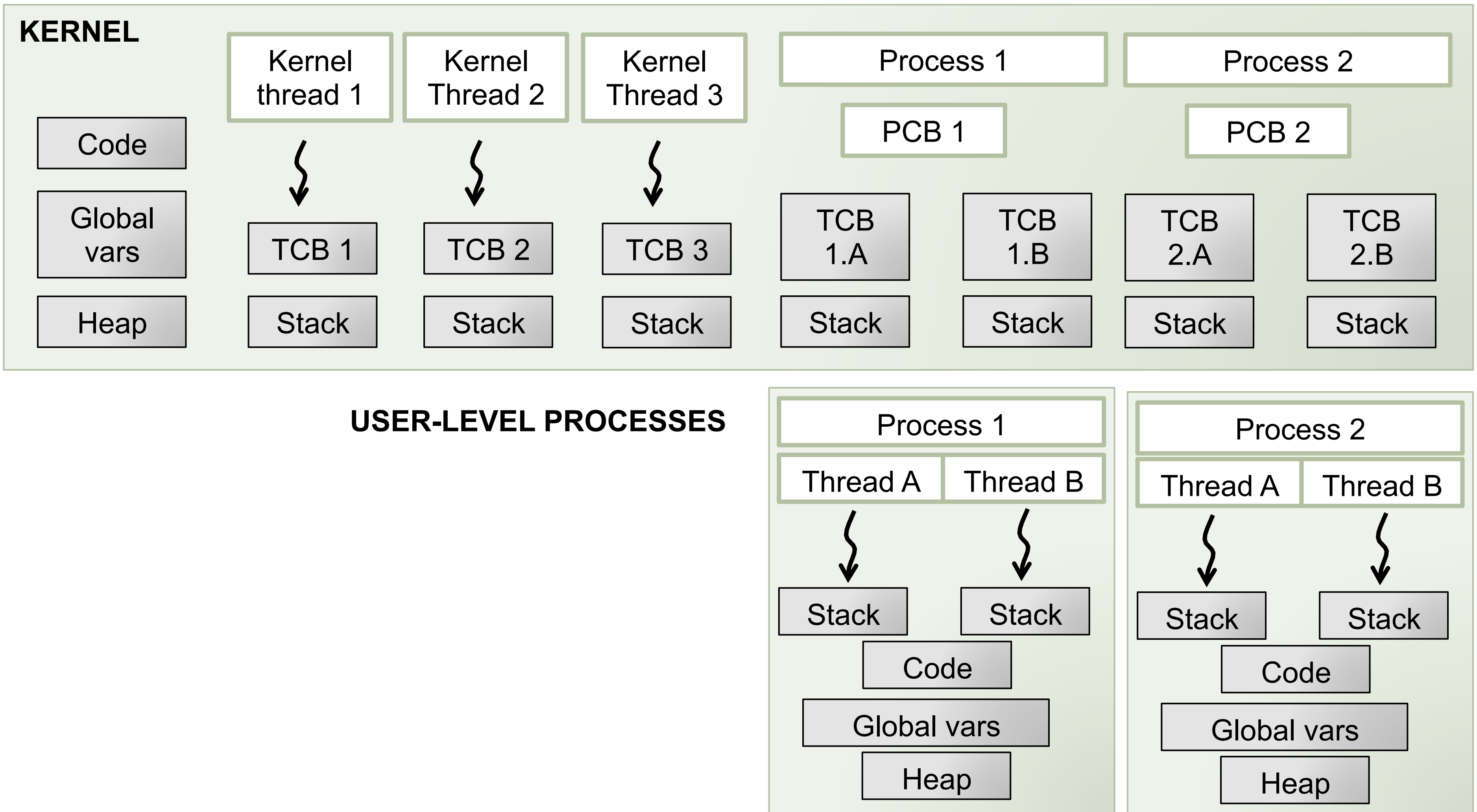
Shared vs. Per-Thread State



Thread Implementation

- Threads can be implemented in any of several ways
  - Multiple user-level threads, inside a UNIX process (early Java)
  - Multiple single-threaded processes (early UNIX)
  - Mixture of single and multi-threaded processes and kernel threads (Linux, MacOS, Windows)
    - To the kernel, a kernel thread and a single threaded user process look quite similar
  - Scheduler activations (Windows)

Multi-threaded kernel and multi-threaded processes



Creating a thread

- thread_create(func, arg)
  - Allocate thread control block (TCB)
  - Allocate stack
  - Build stack frame for base of stack
  - Put func, arg on stack
  - Set PC to stub
  - Put thread on ready list
  - Will run sometime later (maybe right away!)

- stub
  - Run function func with argument arg
  - thread_exit(0)

Thread Switch (in C)

```
void thread_switch(oldThreadTCB, newThreadTCB ) {
  save_state(oldThreadTCB);
  oldThreadTCB->sp = %ESP

  %ESP = newThreadTCB-> sp
  load_state(newThreadTCB);
}
```

Implementing (voluntary) thread context switch

- Disable interrupts
- Get next thread
  - If null? Go back to the original thread
- Switch contexts
  - Change the state of the threads
  - Put current thread in ready queue
  - Call thread_switch
- Enable interrupts

Two threads call yield

Thread 1's instructions	**Thread 2's instructions**	**Processor's instructions**
call thread_yield		call thread_yield
save state to stack		save state to stack
save state to TCB		save state to TCB
choose another thread		choose another thread
load other thread state		load other thread state
	call thread_yield	call thread_yield
	save state to stack	save state to stack
	save state to TCB	save state to TCB
	choose another thread	choose another thread
	load other thread state	load other thread state
return thread_yield		return thread_yield
call thread_yield		call thread_yield
save state to stack		save state to stack
save state to TCB		save state to TCB
choose another thread		choose another thread
load other thread state		load other thread state
	return thread_yield	return thread_yield
	call thread_yield	call thread_yield
	save state to stack	save state to stack
	save state to TCB	save state to TCB
	choose another thread	choose another thread
	load other thread state	load other thread state
return thread_yield		return thread_yield
...	...	...

Thread switch on an interrupt

- Thread switch can occur due to timer or I/O interrupt
  - Tells OS some other thread should run

- Simple version
  - End of interrupt handler calls `thread_switch()`
  - When resumed, return from handler resumes kernel or user thread

- Faster version
  - Interrupt handler returns to saved state in TCB
  - Could be kernel or user thread

Multiple Processors

- Thread switch is no longer sufficient

- Usual approach
  - run on each processor an idle thread

```
void idle_thread() {
  while(1)
    if (need_resched())
          sched();
}
```

- Why?

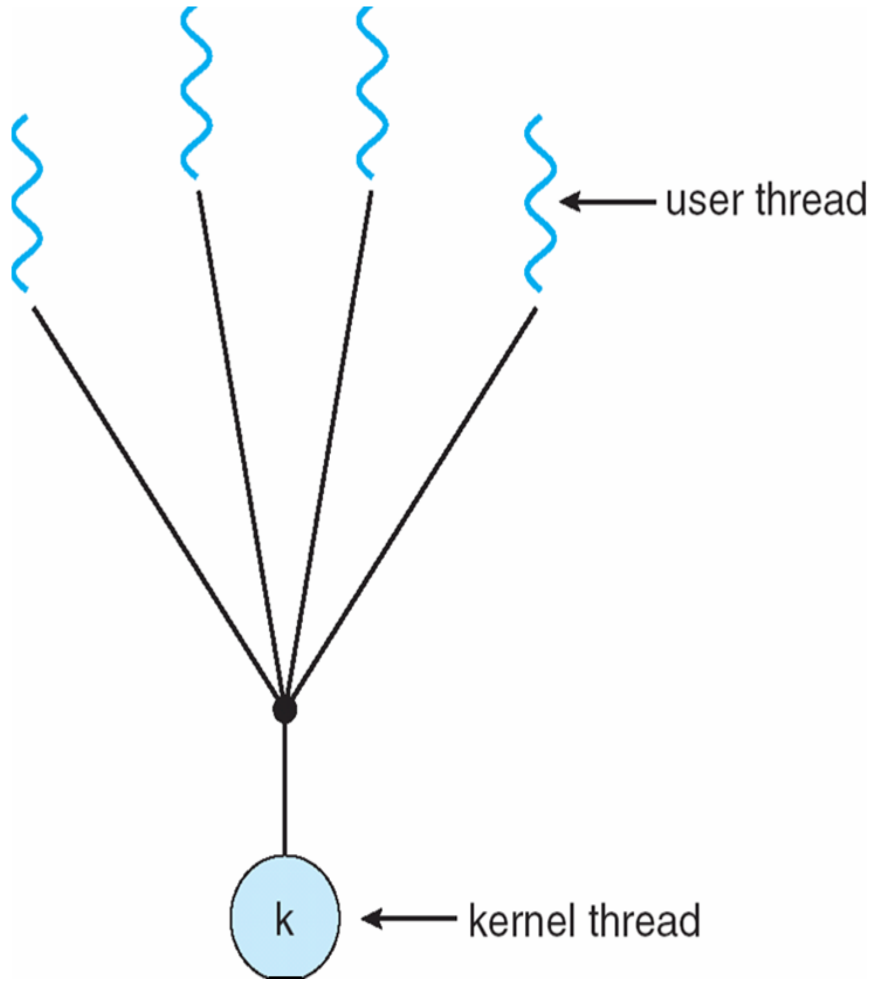
Threads in a Process

- User-level library, within a single-threaded process
  - → Model many-to-one
- Use kernel threads
  - → Model one-to-one
- Use scheduler activations
  - → Model many-to-many
- Use event-driven programming

Many-to-One (N:1)

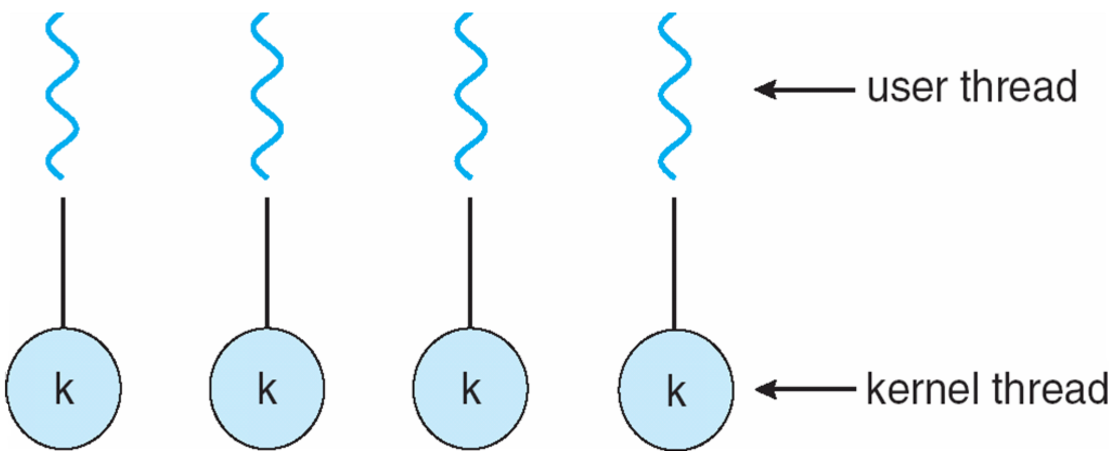
- Many user-level threads mapped to single kernel thread
  - Library does thread context switch
  - Kernel time slices between processes, e.g., on system call I/O

- Examples:
  - Solaris Green Threads
  - GNU Portable Threads
  - Early Java



One-to-One (1:1)

- Each user-level thread maps to kernel thread
  - System calls for thread fork, join, exit (and lock, unlock,…)
  - Kernel does context switching
- Simple, but a lot of transitions between user and kernel mode
- Examples
  - Win32
  - Linux (NPTL)
  - Solaris 9 and later
  - OS X
  - FreeBSD
  - …

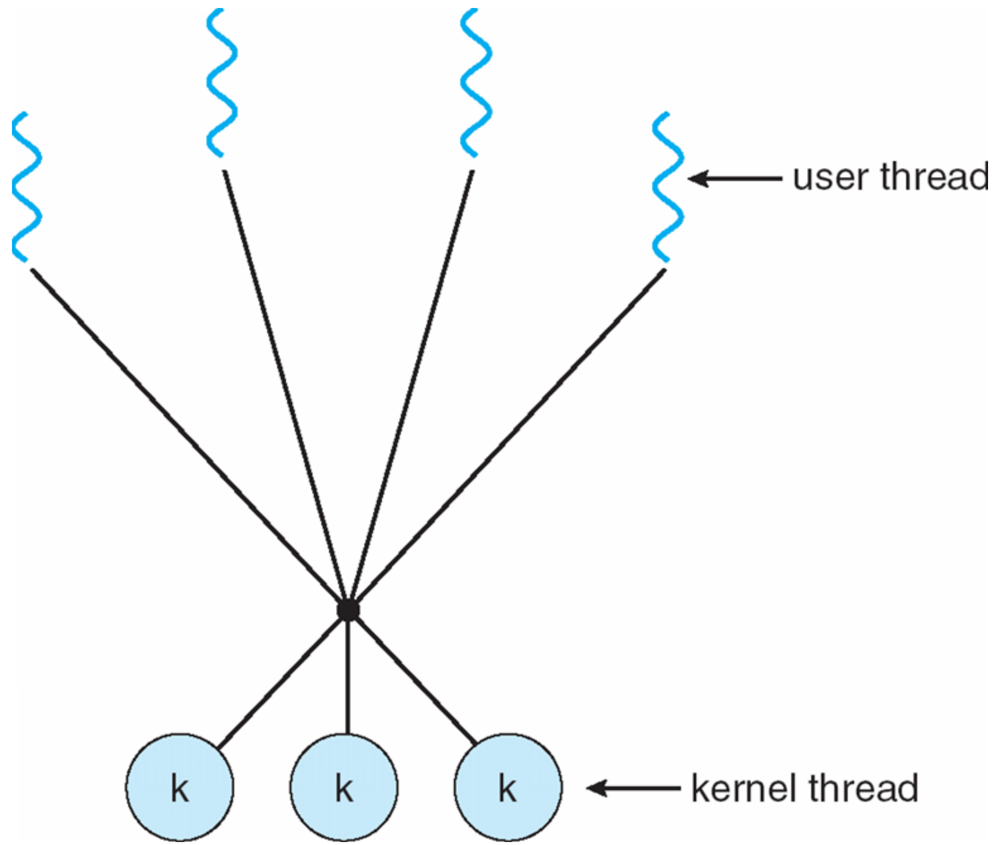


Slides adapted from Tom Anderson's "Operating Systems: Principles and Practice"

Many-to-Many (M:N)

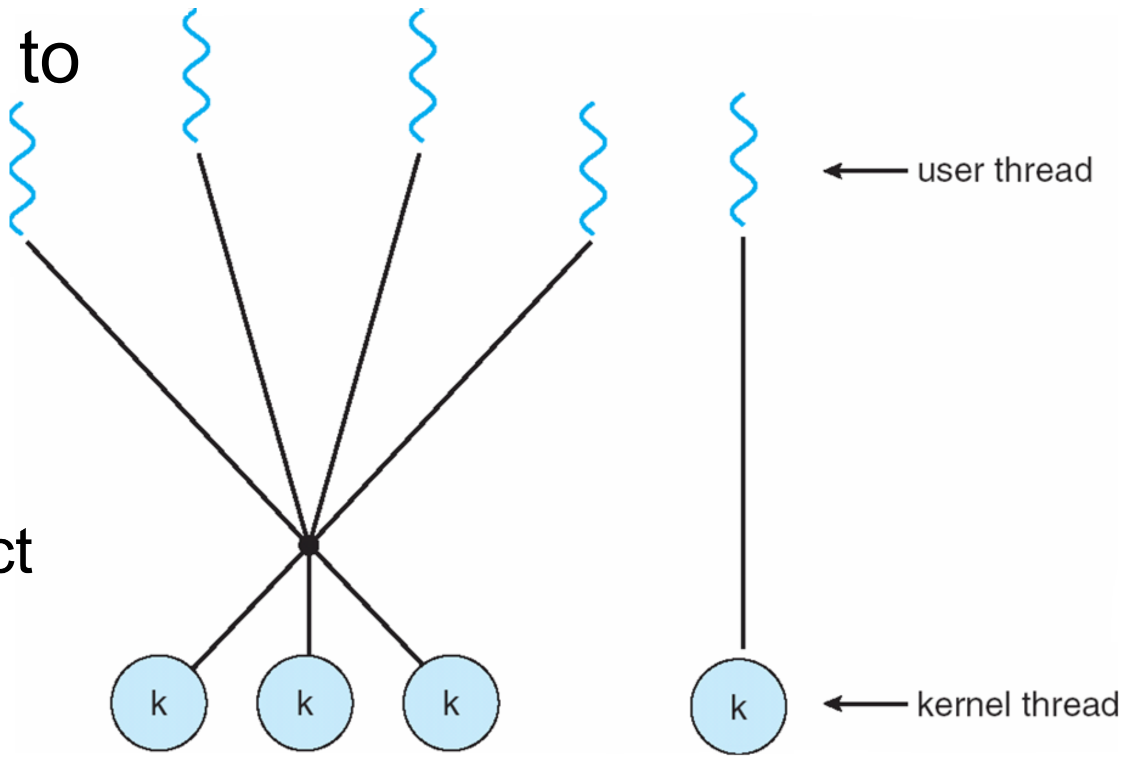
- Allows many user level threads to be mapped to many kernel threads
  - Kernel allocates processors to user-level library
  - Thread library implements context switch
  - System call I/O that blocks triggers upcall

- Examples
  - Solaris prior to version 9
  - Windows NT/2000 with the ThreadFiber package

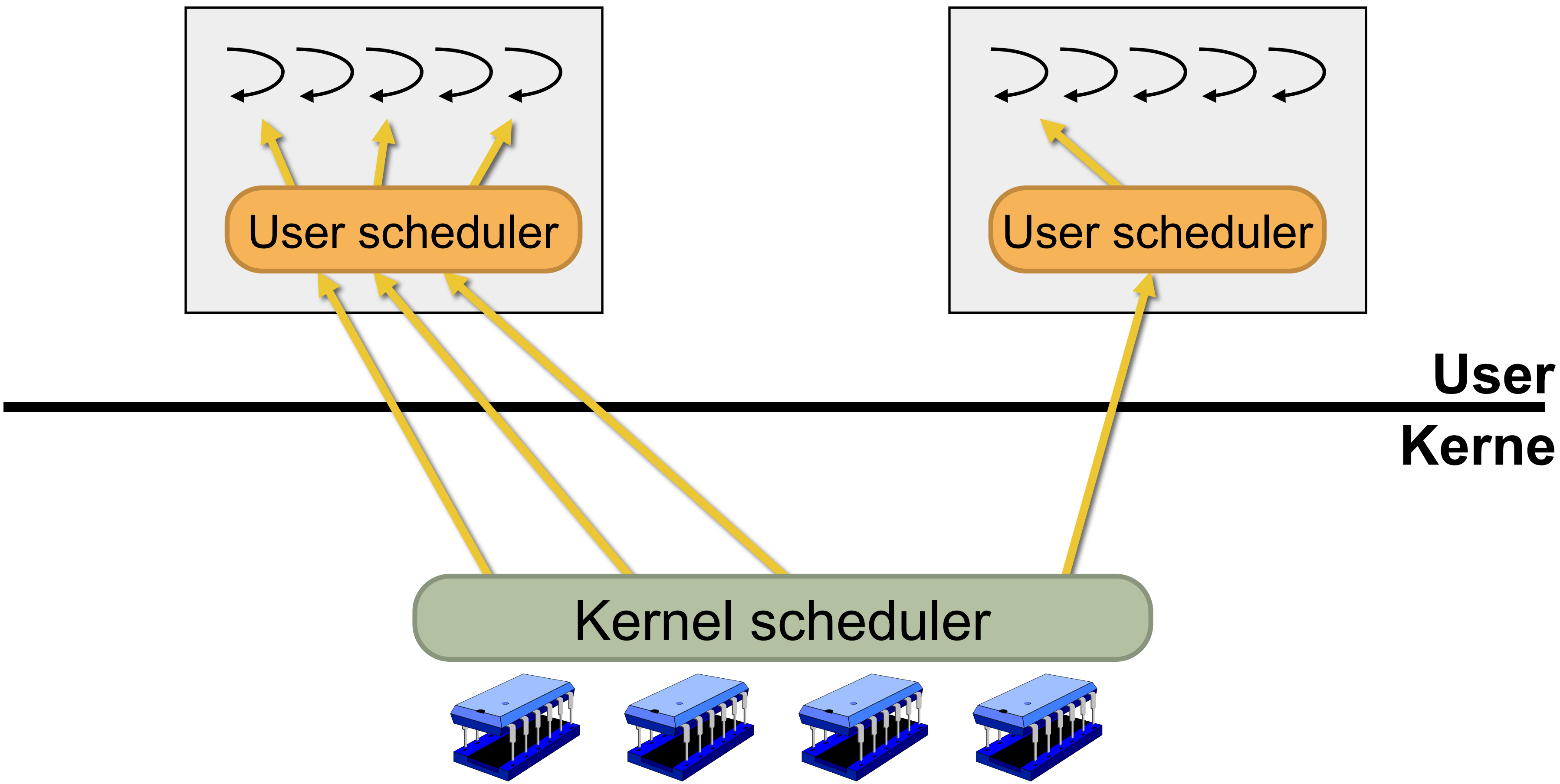


Hybrid

- Similar to M:M, except that it allows a user thread to be bound to kernel thread
- Examples
  - HP-UX
  - Marcel – PM2 project
  - Windows 7

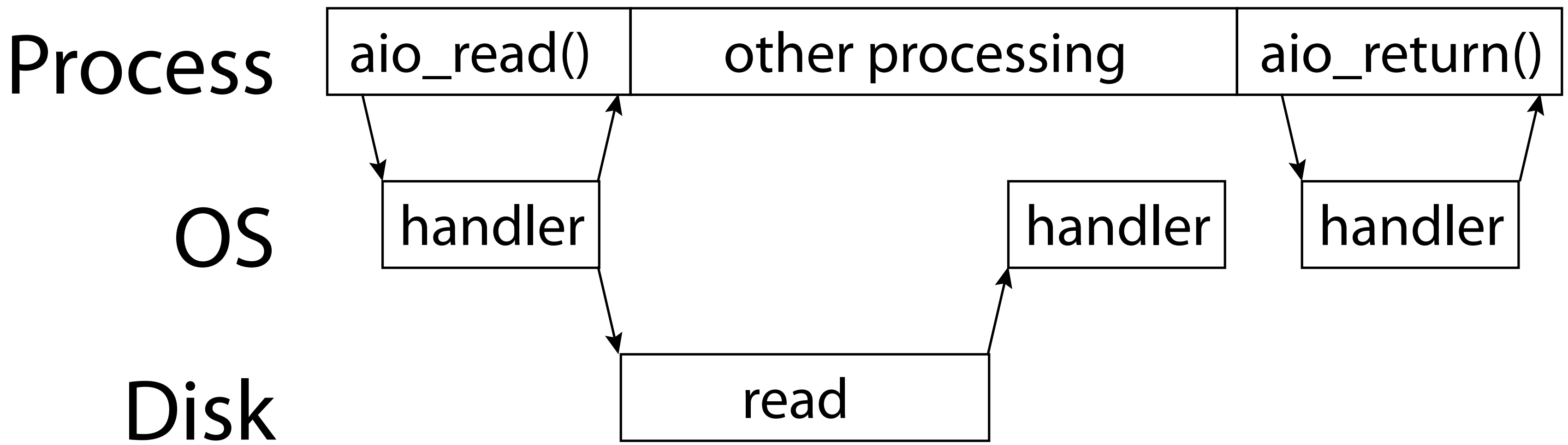


Scheduler Activations



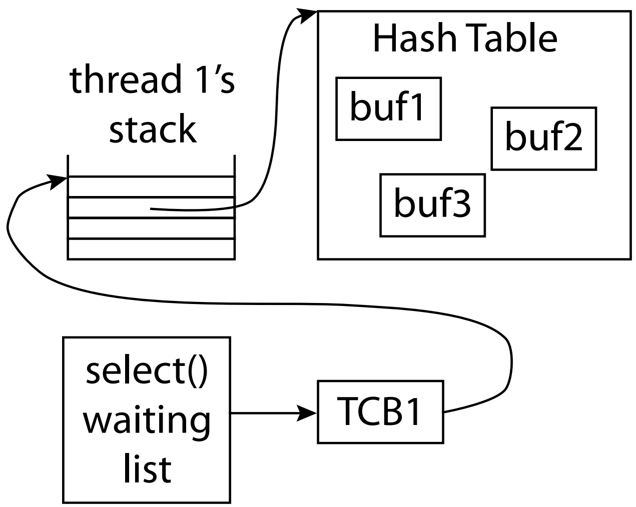
Slides adapted from Tom Anderson's "Operating Systems: Principles and Practice"

Event-Driven Programming



Event-Driven vs Threads

Event-Driven



Threads