From where was this shot taken?

System Calls Threads

Recap

- Processes are represented by the PCB (process control block) data structure in the kernel.
- Processes switch between 3 different states
  - ready, running, and blocked
- A context switch is the mechanism by which we switch from running one process to another
  - requires saving state of running process
  - restoring state of process to which we are switching

How does the kernel (OS) get to run?

- Hardware or software interrupt
  - executes interrupt handling code
  - switches to kernel mode
  - jumps to kernel function to handle the interrupt

- What causes an interrupt?
  - timer interrupt
  - system call
  - error like segfault, divide by 0

Limited Direct Execution

- Direct execution - a process executes its instructions on the CPU without going through a controller
- Limited - user processes cannot execute privileged instructions

Enforcing Restrictions

- Hardware runs in user mode or system mode
- Some instructions are privileged instructions: they can only run in system mode
- On a "system call interrupt", the mode bit is switched to allow privileged instructions to occur

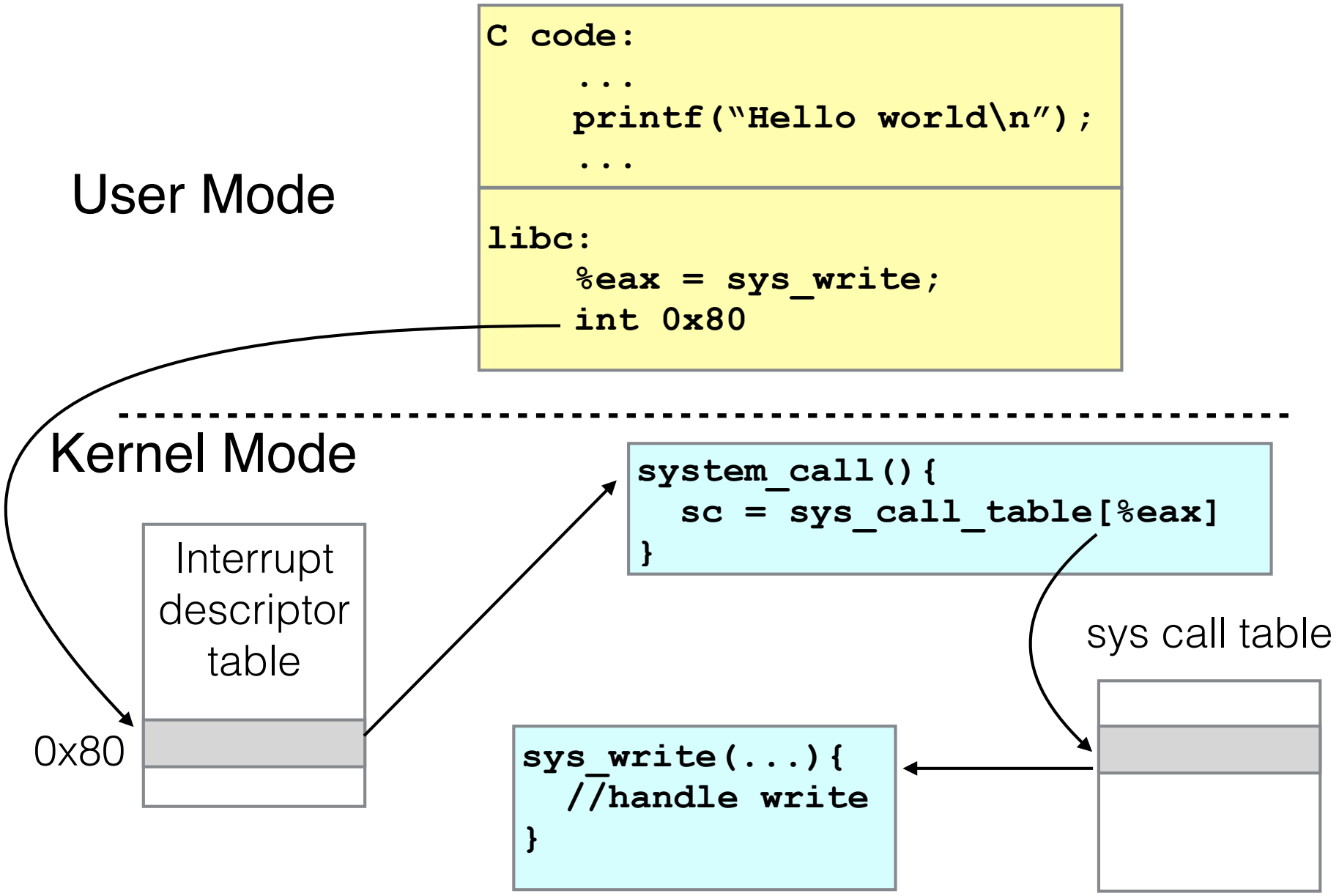
Privileged instructions

- Access I/O device
  - Poll for IO, perform DMA (Direct Memory Access), catch hardware interrupt
- Manipulate memory management
  - Set up page tables, load/flush the TLB and CPU caches (we'll see this later), etc.
- Configure various "mode bits"
  - Interrupt priority level, software trap vectors, etc.
- Call halt instruction
  - Put CPU into low-power or idle state until next interrupt
- These are enforced by the CPU hardware itself
  - Reminder: CPU has at least 2 protection levels: Kernel and user mode
  - CPU checks current protection level on each instruction!
  - What happens if user program tries to execute a privileged instruction?

System Call Interface

- User program calls C library function with arguments
- C library function passes arguments to OS
  - Includes a system call identifier!
- Executes special instruction (x86: INT) to trap to system mode
  - Interrupt/trap vector transfers control to a handler routine
- Syscall handler figures out which system call is needed and calls a routine for that operation
- How does this differ from a normal C language function call? Why is it done this way?

Example: Linux `write` system call



System Call Operation

- Kernel must verify arguments that it is passed
  - Why?
- A fixed number of arguments can be passed in registers
  - Often pass the address of a user buffer containing data (e.g., for write())
  - Kernel must copy data from user space into its own buffers
- Result of system call is returned in register EAX

System Calls in Linux

- Can invoke any system call from userspace using the syscall() function

```
syscall(syscall_no, arg1, arg2, arg3, ..)
```

- E.g.,
  - `const char msg[] = "Hello World!";`
  - `syscall(4, STDOUT_FILENO, msg, sizeof(msg)-1);`
  - Equivalent to: `write(STDOUT_FILENO, msg, sizeof(msg)-1);`

- Tracing system calls:
  - Powerful mechanism to trace system call execution for an application
  - Use the `strace` command
  - The `ptrace()` system call is used to implement `strace` (also used by gdb)
  - Library calls can be traced using `ltrace` command

System Call Dispatch

- Why do we need a system call table?
  - How would you get to the right routine?
  - If-then-else for each system call number?
  - Too inefficient!
- A system call is identified by a unique number
  - The system call number is passed into register %eax
  - Offers an index into an array of function pointers: the system call table!
- System call table: **`sys_call_table[__NR_syscalls]`** (approximately 300)
- See all syscalls in VM: /usr/src/linux-source-2.6.32/arch/x86/kernel/syscall_table_32.S

System call dispatch

1. Kernel assigns each system call type a system call number

2. Kernel initializes system call table, mapping each system call number to a function implementing that system call

3. User process sets up system call number and arguments

4. User process runs `int N` (on Linux, N=0x80)

5. Hardware switches to kernel mode and invokes kernel's interrupt handler for X (interrupt dispatch)

6. Kernel looks up syscall table using system call number

7. Kernel invokes the corresponding function

8. Kernel returns by running iret (interrupt return)

Passing System Call Parameters

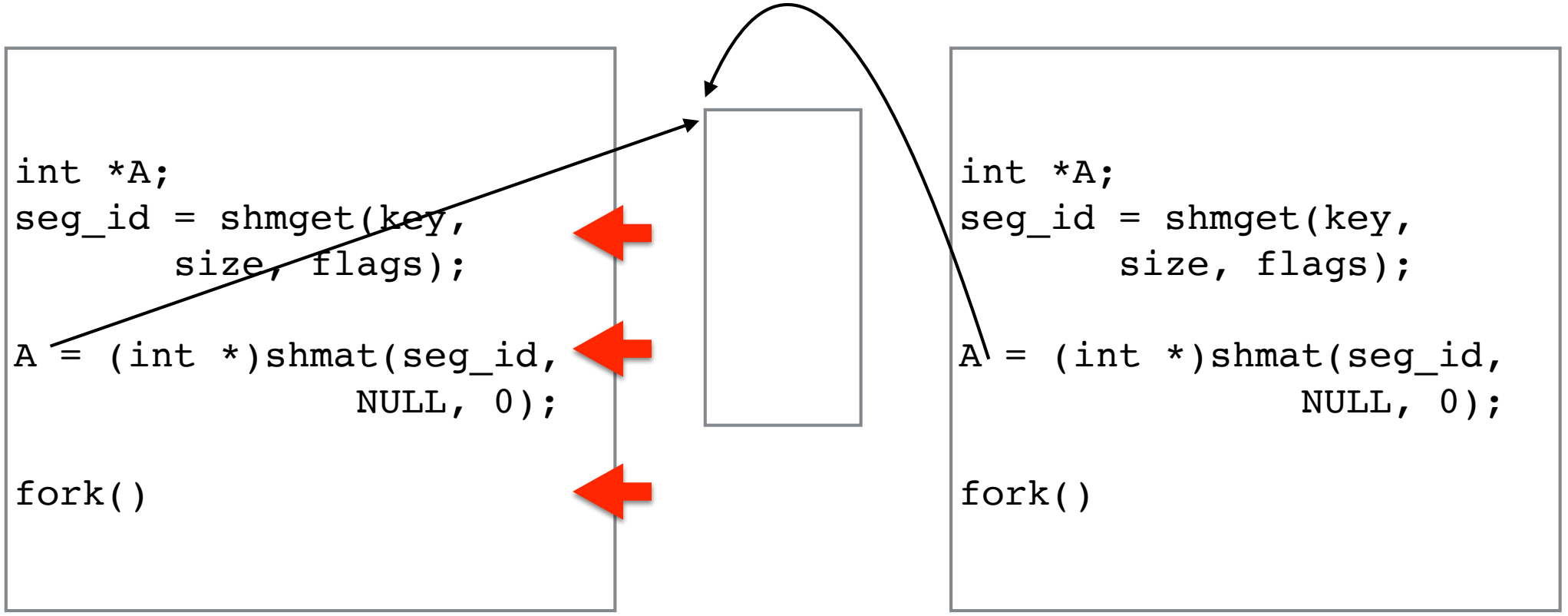
- The first parameter is always the syscall number
  - Stored in eax
- Can pass up to 6 parameters:
  - ebx, ecx, edx, esi, edi, ebp
- If more than 6 parameters are needed, package the rest in a struct and pass a pointer to it as the 6th parameter
- Problem: must validate user pointers. Why? How?
- Solution: safe functions to access user pointers: copy_from_user(), copy_to_user(), etc.

Processes and Threads

Interprocess communication

- How do processes communicate?
  - signals
  - pipes
  - sockets
  - files
- Wouldn't it be nice if cooperating processes could just share memory?

Processes can share memory?



- `shmget()`: System call to tell OS to allocate a shared memory region
- `shmat()`: map shared memory to local address
- `mmap()` is another approach for creating shared memory regions

Parallel Programs

- To execute a web server, or any parallel program using fork(), we need to
  - Create several processes that execute in parallel
  - Cause each to map to the same address space to share data
  - They are all part of the same computation
  - Have the OS schedule these processes in parallel (logically or physically)

- This situation is very inefficient
  - Space: PCB, page tables, etc.
  - Time: create data structures, fork and copy addr space, etc.
  - Inter-process communication (IPC): extra work is needed to share and communicate across isolated processes

From Processes to Threads

Recall that process is:

- Address space
  - Code
  - Heap
- Execution state
  - program counter
  - stack and stack pointer
- OS resources
  - open file table, etc.

Big idea

- Separate the address space from the execution state
- Then multiple “threads of execution” can execute in a single address space.

Why?

- Sharing:
  - threads can solve a single problem concurrently and can easily share code, heap, and global variables
- Lighter weight:
  - faster to create and destroy
  - potentially faster context switch times
- Concurrent programming performance gains:
  - Overlapping computation and I/O

What is a Thread?

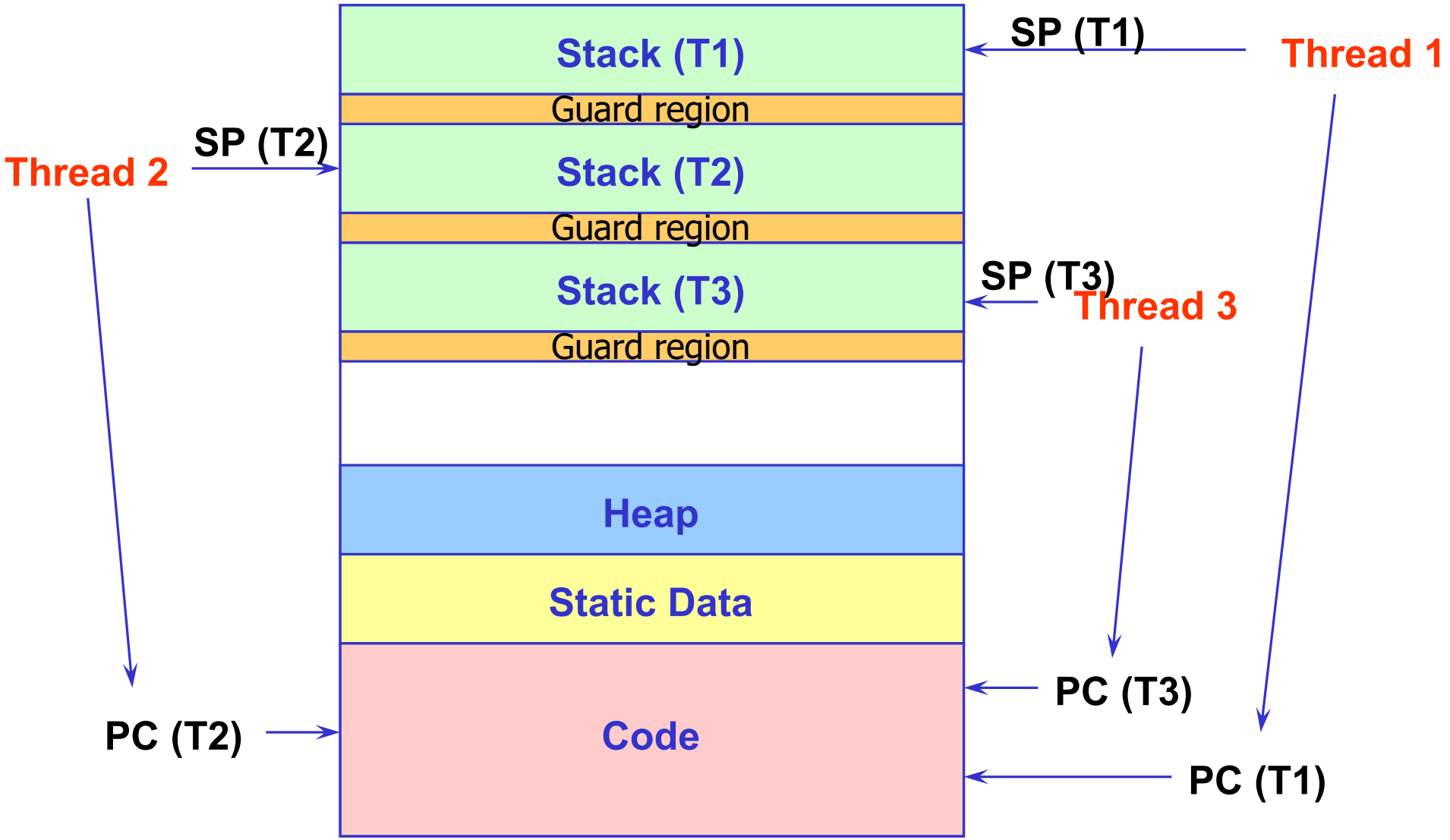
- A thread is a single *control flow through a program*
  - What is a “control flow?
  - How is it represented?
- A program with multiple control flows is *multithreaded*
  - OS must interact with multiple running programs, so it is naturally multithreaded

```
int foo() {
  …
}

void bar() {
  …
}

main() {
  x = foo();
  if (x > 0) {
     bar();
  } else {
     printf(“Error\n”);
  }
}
```

Multithreaded Process Address Space



Kernel-Level Threads

- Modern OSs have taken the execution aspect of a process and separated it out into a thread abstraction
  - To make concurrency cheaper
- The OS now manages threads and processes
  - All thread operations are implemented in the kernel
  - The OS schedules all of the threads in the system
- Called kernel-level threads or lightweight processes
  - Linux: tasks NT: threads Solaris: lightweight processes

Kernel Thread Limitations

- Kernel-level threads make concurrency much cheaper than processes
  - Less state to allocate and initialize
- However, for fine-grained concurrency, kernel-level threads still suffer from too much overhead
  - Thread operations still require system calls
    - Ideally, want thread operations to be as fast as a procedure call
  - Kernel-level threads have to be general to support the needs of all programmers, languages, runtimes, etc.
- For fine-grained concurrency, need even “cheaper” threads

User-Level Threads

- To make threads cheap and fast, they need to be implemented at user level
  - Kernel-level threads are managed by the OS
  - User-level threads are managed entirely by the run-time system (user-level library)

- User-level threads are small and fast
  - A thread is simply represented by a PC, registers, stack, and small thread control block (TCB)
  - Creating a new thread, switching between threads, and synchronizing threads are done via procedure call (no kernel involvement)
  - User-level thread operations are up to 100x faster than kernel threads
    - But this depends on the quality of both implementations!

User Level Thread Limitations

- But, user-level threads are not a perfect solution
  - As with everything else, they are a tradeoff
- User-level threads are invisible to the OS
  - They are not well integrated with the OS
- As a result, the OS can make poor decisions
  - Scheduling a process with idle threads
  - Blocking a process whose thread initiated an I/O, even though the process has other threads that can execute
  - De-scheduling a process with a thread holding a lock
- Solving this requires communication between the kernel and the user-level thread manager

POSIX Threads

- Standardized C language threads programming API
- Known as pthreads
- Specifies interface, not implementation!
  - Implementation may be kernel-level threads, user-level threads, or hybrid
  - Linux pthreads implementation uses kernel-level threads
- Excellent tutorial:
  - https://computing.llnl.gov/tutorials/pthreads/

Examples and take-aways

- sequential vs. parallel, pthreads vs. processes ...
- Bottomline – in general:
  - processes are more heavy-weight than threads and have a higher startup/shutdown cost (..not by much in Linux though)
  - processes are safer and more secure (each process has its own address space)
    - a thread crash takes down all other threads
    - a thread's buffer overrun creates security problem for all
- Which one to use? Depends on application type!