What is a thread?

Bringing It Together

- OS has three hats: What are they?
  - Processes help with one? two? three? of those hats
- OS protects itself and others through dual-mode execution
  - Which relates to processes how?
- So then we got to the illusionist part: CPU Scheduling
  - discussed the possible policies the scheduler may use to choose the next process (or thread!) to run
  - criteria we use to evaluate policies
    - throughput, turnaround time, response time, CPU utilization, waiting time (and one more! What was it?)
  - FIFO, Round Robin, SJF, Multilevel Feedback Queues

Processes: What We Think We Know

- A process is the abstraction used by the OS to manage resources and provide protection
- A process defines an address space
  - Identifies all addresses that may be touched by the program
- A process has a single *thread of control* that executes instructions sequentially
  - Creates a single sequential execution stream

What if we decouple the thread of control information from the process?

Threads

- A *thread* represents an abstract entity that executes a sequence of instructions
  - Short for “Thread of Control”
  - Defines a single sequential execution stream within a process
- A thread is bound to a single process
- Each process may have multiple threads of control
  - *Must* have one
  - Virtualizes the processor

Why Threads?

Why Threads?

- Programmers create *multi-threaded* programs to:
  - Better represent the structure of the tasks
    - We think linearly
    - But the world is concurrent!
  - Improve performance
    - One thread can perform computation while another waits for I/O
    - Threads may be scheduled across different processors in a multi-processor architecture
- First, concrete examples of usefulness, then implementation, then interaction (or not) with the OS (which is based on thread type)

The Case for Threads: Web Servers

Consider a web server that performs these actions:

```
while()
    get network message (URL) from client
    get URL data from disk
    compose response
    send response
```

How well does this web server perform?

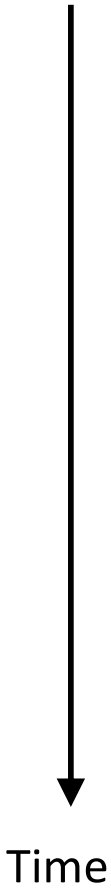
The Case for Threads: Web Servers

Consider a web server that performs these actions:

- Create a number of threads, and for each thread do:
  - get network message (URL) from client
  - get URL data from disk
  - compose response
  - send response

How well does this web server perform?

Overlapping Requests (Concurrency)



Thread One, Request One

- get network message (URL) from client
- get URL data from disk

---disk access latency---

- send data over network

Thread Two, Request Two

- get network message (URL) from client
- get URL data from disk

---disk access latency---

- send data over network

=> Total time is less than request 1 + request 2

The Case for Threads: Arrays

Consider the following code fragment:

```
for(k = 0; k < n; k++)
    a[k] = b[k] * c[k] + d[k] * e[k];
```

Is there a missed opportunity?

Question

Switching between which of these entities is the most expensive?

A. Processes
B. Threads

Threads are Lightweight

- Creating a thread is cheaper than creating a process
- Communication between threads is easier than between processes
  - Processes must set up a shared resource or pass messages or signals
- Context switching between threads is cheaper (same address space)

Using Threads

Programmer's View

```
void thread_function(int arg0, int arg1, …) {…}


main() {
   …
   tid = thread_create(thread_function, arg0, arg1, …);
   …
}
```

At the point `thread_create()` is called:

- execution continues with the original thread in `main` function, and
- execution starts at `thread_function()` in new thread

in parallel (concurrently).

Programmer's View: Array Example

How can this code take advantage of 2 threads?

```
for(k = 0; k < n; k++)
  a[k] = b[k] * c[k] + d[k] * e[k];
```

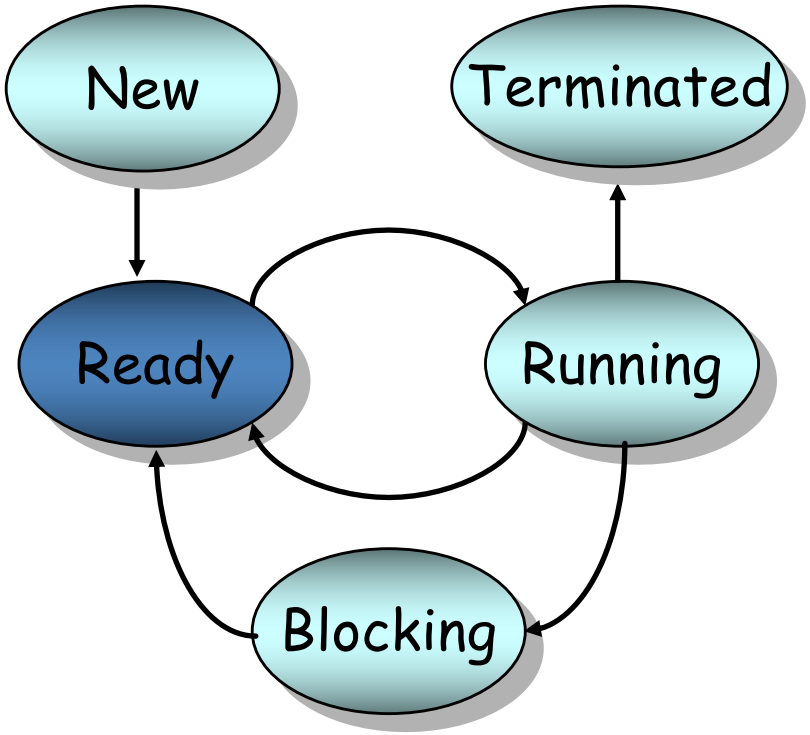
Rewrite this code fragment as:

```
do_mult(start, end) {            /*thread function*/
  for(k = start; k < end; k++)
   a[k] = b[k] * c[k] + d[k] * e[k];
}
main() {
   /*args are thread function name and then its args*/
   thread_create(do_mult, 0, n/2);
   thread_create(do_mult, n/2, n);
```

A Closer Look

Threads: A Closer Look

Threads (just like processes) go through a sequence of *new*, *ready*, *running*, *blocking*, and *terminated* states



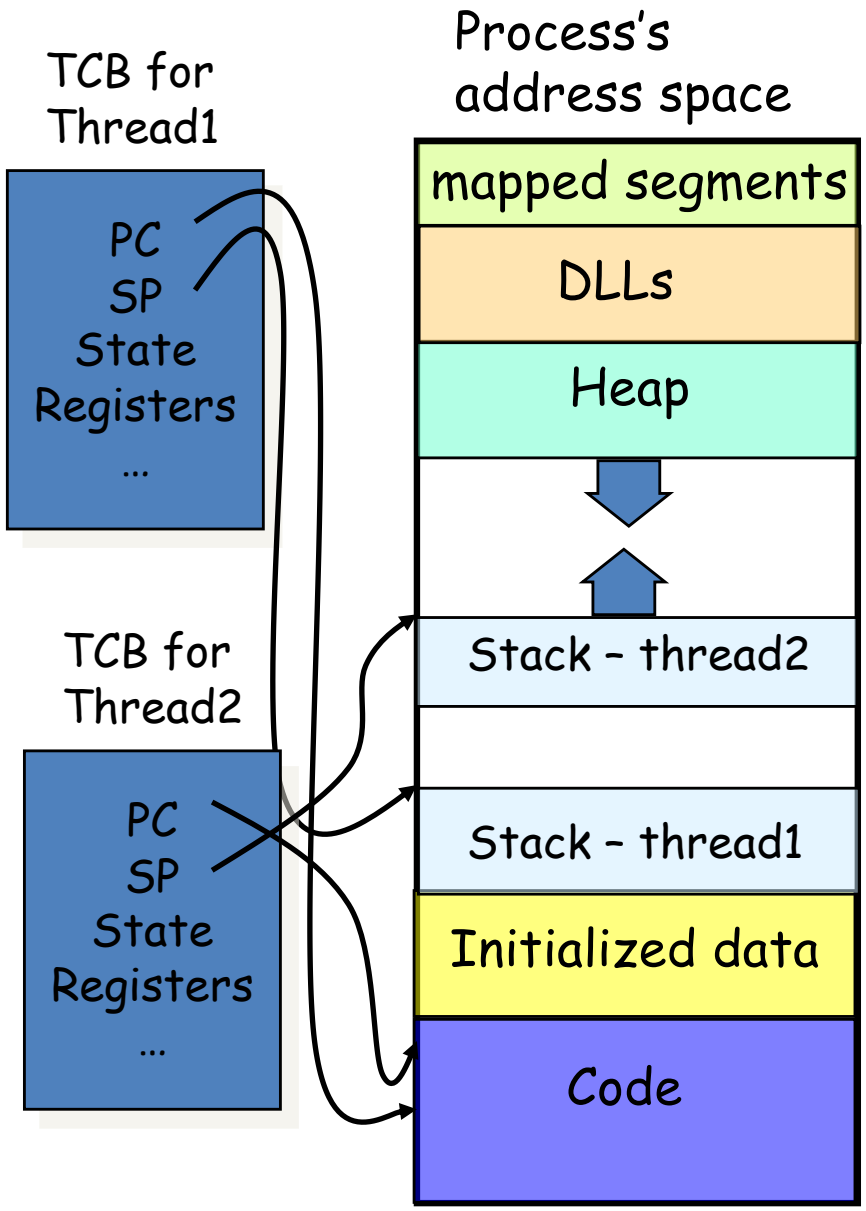
Threads and the Address Space

- Processes *define* an address space; threads of the same process *share* the address space
  - All process data can be accessed by any thread
    - Particularly global data
    - Heap is also shared (*What about pointers into the heap?*)
- Each thread has:
  - Its own stack
    - BUT there is no protection…So any thread can modify another thread's stack

      (This is usually a bug)
  - Exclusive use of the CPU registers while it is executing
    - When a thread is pre-empted, its register values are saved as part of its state
    - the new thread gets to use the registers!

Metadata Structures

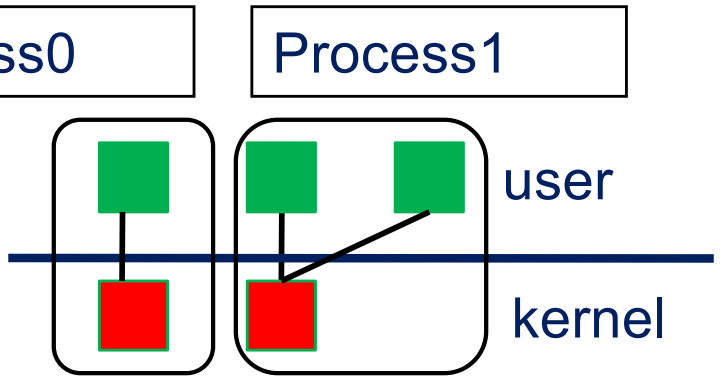
- Process Control Block (PCB) contains process-specific information
  - Owner, PID, heap pointer, priority, active thread, and pointers to thread information
- Thread Control Block (TCB) contains thread-specific information
  - Stack pointer, PC, thread state (running, …), register values, a pointer to PCB, …



Threads vs. Processes

Thread Types

User-Level Threads



- A *user-level thread* is a thread the OS does **not** know about
- OS **only** schedules the process **not** the threads within a process
- Programmer uses a *thread library* to manage threads (create, delete, synchronize, and schedule)
  - User-level code can define scheduling policy
  - Threads *yield* to other threads or voluntarily give up the processor
- Switching threads does not involve a context switch

User-Level Threads: ~~Context~~ Switches

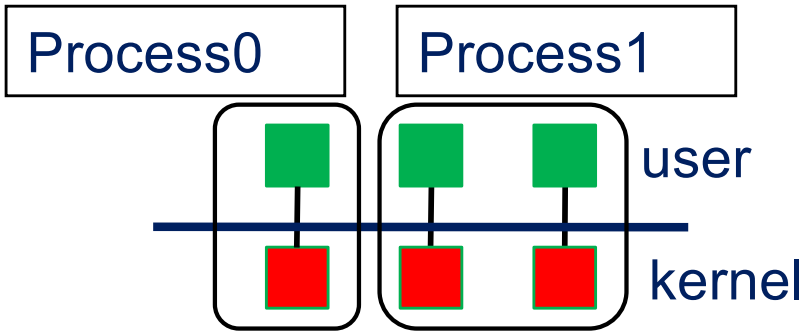
between threads of the same process

Similar to processes *except entirely in user space*:

- Thread is running
- Thread is interrupted by ~~an interrupt~~ *a signal or voluntarily yields*
- ~~Switch to kernel~~
- ~~Kernel~~ *Library* code saves thread state (to TCB)
- ~~Kernel~~ *Library* code chooses new thread to run
- ~~Kernel~~ *Library* code loads its state (from TCB)
- Thread is running

What happens if the thread blocks?

Kernel-Level Threads



- A *kernel-level thread* is a thread that the OS knows about
  - Every process has at least one kernel-level thread
- Kernel manages and schedules threads (as well as processes)
  - System calls used to create, destroy, and synchronize threads
- Switching between kernel-level threads of the same process requires a small context switch
  - Values of registers, program counter, and stack counter must be switched
  - Memory management information remains since threads share an address space

Kernel-Level Threads: Context Switches

between threads of the same process

Similar to processes:

- Thread is running
- Thread blocks, is interrupted, *or voluntarily yields*
- Mode switch to kernel mode
- Kernel code saves thread state (to TCB)
- Kernel code chooses new thread to run
- Kernel code loads its state (from TCB)
- Mode switch to user mode
- Thread is running

The Power of Kernel Threads

- I/O: the OS can choose another thread in the same process when a thread does I/O
  - Non-blocking calls are good in theory, but difficult to program in practice
- Kernel-level threads can exploit parallelism
  - Different processors of a symmetric multiprocessor
  - Different cores on a multicore CPU
- Used by systems: Linux, Solaris, Windows, pthreads (usually)
- Also used by recent implementations of Java

Kernel-Level vs. User-Level

One Abstraction, Many Flavors

- Single-threaded processes
  - What we have been assuming
  - Each has exactly one kernel-level thread
  - Add protection
- Multi-threaded processes with user-level threads
  - Threads are created in user-space
  - Have exactly one kernel-level thread
  - Thread management through procedure calls
  - Scheduled by user-space scheduler
  - TCBs in user-space ready list
- Multi-threaded processes with kernel-level threads
  - Threads are created by the OS and thus the OS knows they exist
    - process requests the threads
  - Thread management through system calls
  - TCBs & PCBs on in-kernel ready list
  - Scheduled by OS
- In-kernel threads (New!)
  - Threads that are part of the OS (init, idle)

Note that kernel-level and user-level threads are UNRELATED to kernel-mode and user-mode execution.

One More Flavor: Independent vs. Cooperating Threads

- Independent threads have no shared state with other threads
  - Simple to implement
  - Deterministic
  - Reproducible
  - Scheduling order doesn't matter
- Cooperating threads share state
  - Non-deterministic
  - Non-reproducible
  - Give us concurrency!