CSE 120
Operating Systems Principles

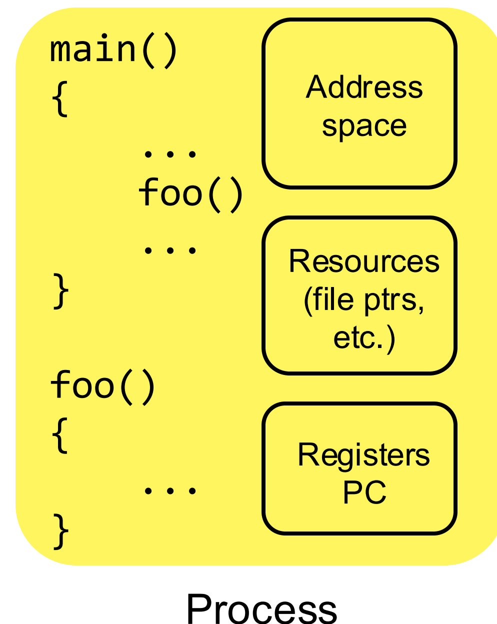
Spring 2025

Lecture 4: Threads

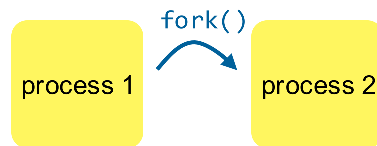
Amy Ousterhout

Processes

- Process: abstraction for a running program
- Recall that a process includes many things
  - An address space
  - OS resources and accounting information
  - Execution state
- Creating a new process is slow
  - Recall 833 LOC for `struct task_struct` in Linux
  - Must create and initialize many data structures
- Communicating between processes is also slow
  - Processes are supposed to be isolated
  - Communication is mediated by the OS



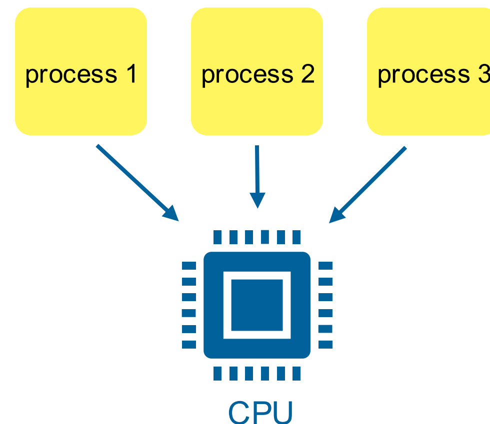
Communication Between Processes



- At process creation time
  - Parents get one chance to pass information via `fork()`
- OS provides mechanisms for communication
  - Called Inter-Process Communication (IPC)
  - Message passing: explicit communication via `send()`/`receive()` system calls
  - Files: `read()`/`write()` system calls
  - Shared memory:
    - » Multiple processes read/write same physical portion of memory
    - » System call to allocate the shared region (e.g., `shm_open()`)
- IPC is typically expensive due to system calls

Concurrent Programs

- Concurrency: multiple tasks in progress at once
- Applications benefit from executing several tasks concurrently
  - Web server – handle multiple requests simultaneously
  - Multicore – utilize multiple cores with one application
  - Overlapping I/O – perform multiple I/O operations in parallel
- We can do this using multiple processes…
  - Create several processes (e.g., with `fork()`)
  - Set up a shared memory region between them
- But this is very inefficient
  - Space: PCBs, memory-management state (page tables)
  - Time: create data structures, fork and copy address space



Rethinking Processes

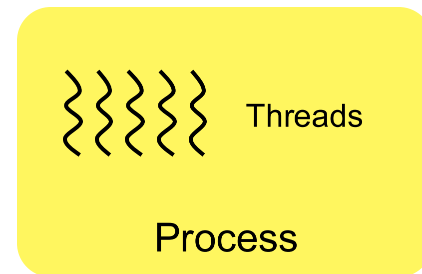
- What is similar in these cooperating processes?
  - They all share the same code and data (address space)
  - They all share the same resources (files, sockets, etc.)
- What don't they share?
  - Each has its own execution state: PC, SP, registers
- Key idea: separate the concept of a process from its execution state
  - Process: address space, privileges, resources, etc.
  - Execution state: PC, SP, registers
- Execution state is called a thread

Today's Outline

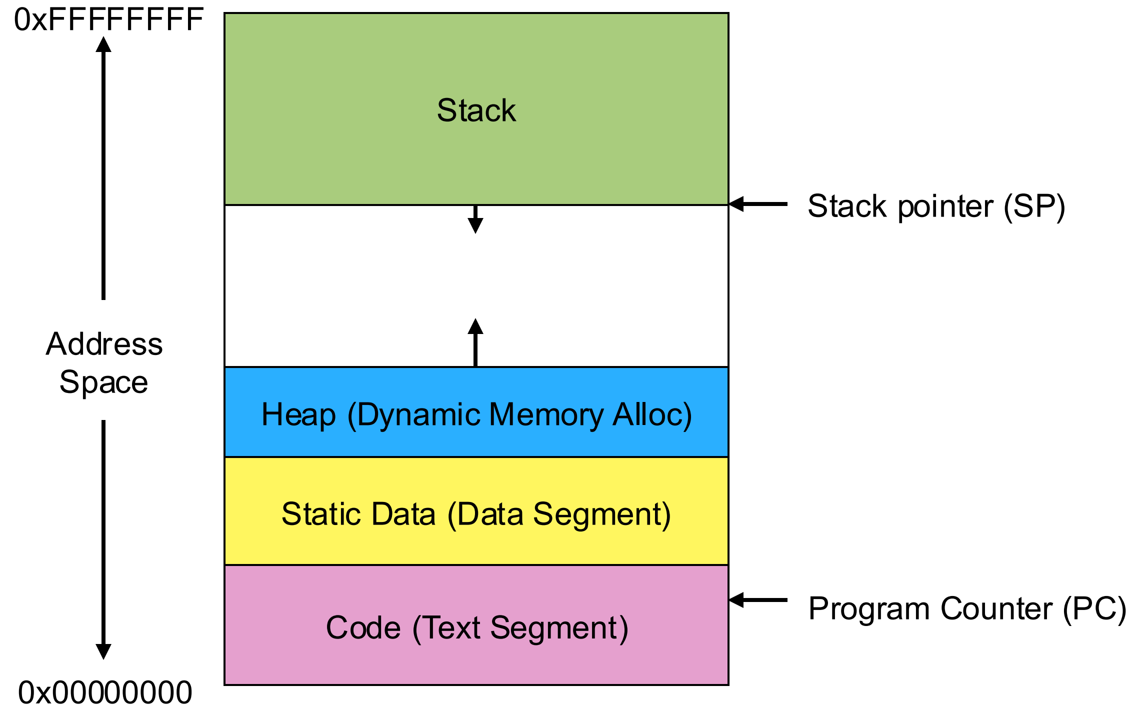
- Threads
  - What is a thread vs. a process?
- Thread scheduling
  - How should we schedule and execute threads?
- Different kinds of threads
  - Should the OS be aware of threads?

Threads

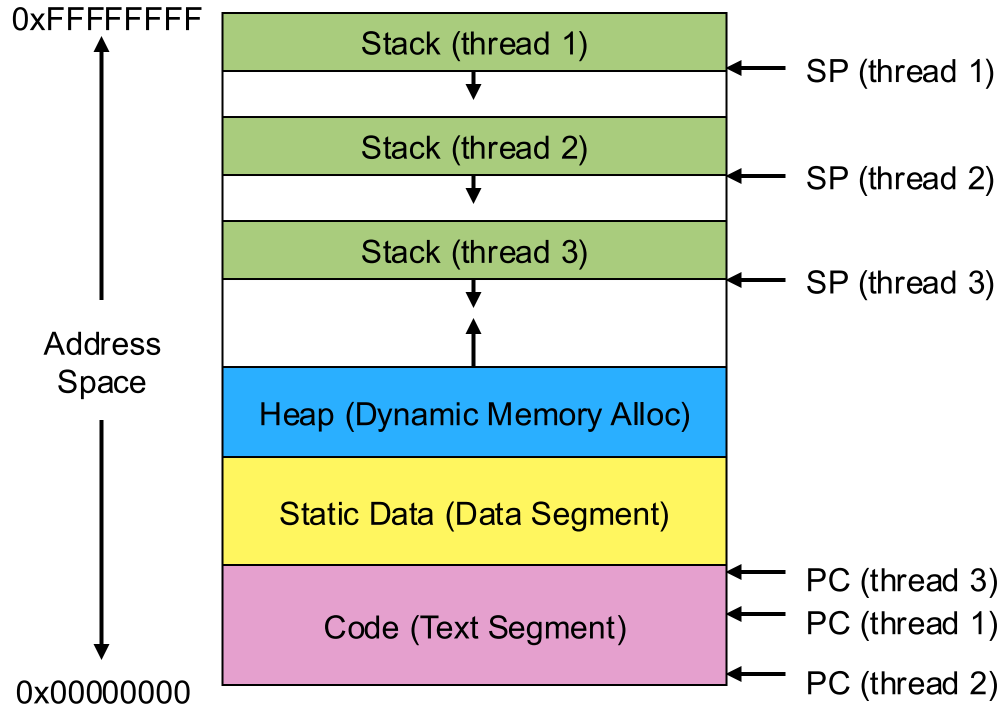
- Modern OSes (Windows, Linux, OS X) separate the concepts of processes and threads
  - Process: address space and resources
  - Thread: a sequential execution stream within a process (PC, SP, registers)
- Each thread is bound to a single process
  - But a process can have multiple threads
- Threads become the basic unit of scheduling
  - Processes are now the containers in which threads execute



Basic Process Address Space

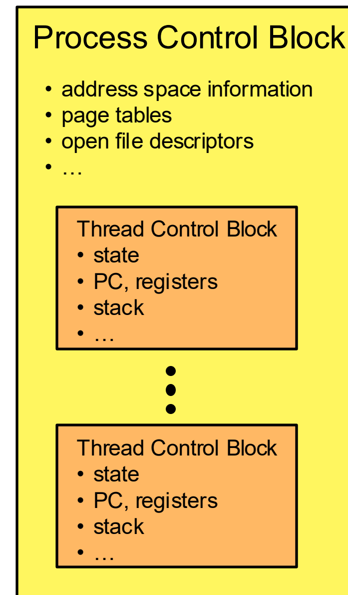


Basic Process Address Space



Thread Control Blocks

- Each Process Control Block contains two kinds of information:
  - Shared information
    - » Memory: code/data/heap segments, page tables
    - » I/O and files: open file descriptors
  - Per-thread information (in Thread Control Blocks)
    - » State (ready, running, or waiting)
    - » PC, registers
    - » Execution stack

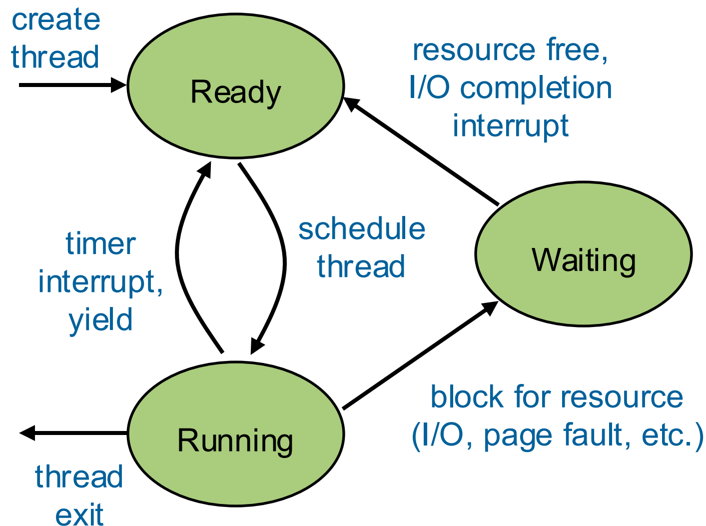


Today’s Outline

- Threads
  - What is a thread vs. a process?
- Thread scheduling
  - How should we schedule and execute threads?
- Different kinds of threads
  - Should the OS be aware of threads?

Thread State Graph

- Each thread has its own execution state (ready, running, waiting, etc.)
  - Indicates what it is currently doing



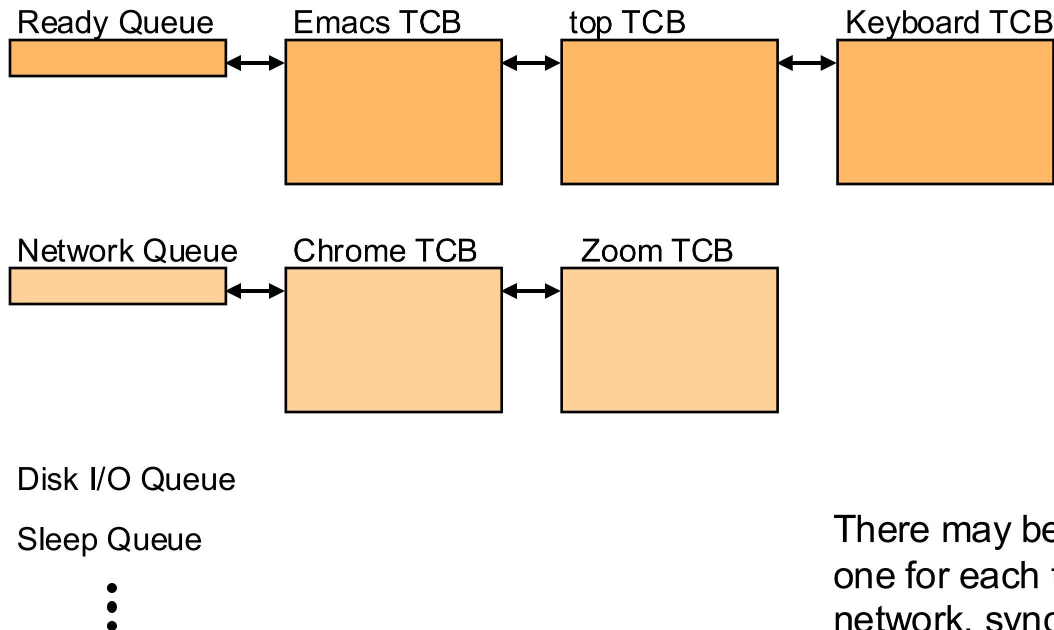
TCBs and Hardware State

- While a thread is running, its hardware state (PC, SP, regs, are in the CPU)
  - Hardware registers contain the current values
- When the OS stops running a thread, it saves the registers into the thread's TCB
- When the OS resumes running a thread, it loads the registers from the values store that thread's TCB
- Context switch – the process of changing the CPU hardware state from one thread to another
  - As often as every millisecond!

Thread Queues

- How does the OS keep track of threads?
- The OS maintains a collection of queues
  - Ready queue – threads that are ready to run
  - Waiting queues
- Each TCB is queued on a state queue according to its current state
  - When a thread changes state, the OS unlinks its TCB from one queue and links it into another

Thread Queues



There may be many wait queues, one for each type of wait (disk, timer, network, synchronization, etc.)

Thread Scheduling

- When should we stop running one thread and switch to another?
- Is this voluntary or involuntary?
  - Preemptive vs. non-preemptive scheduling

Non-Preemptive Scheduling

- Threads voluntarily give up the CPU with `yield()`
- What is the output of running these two threads?

Ping Thread

```
while (1) {
  printf("ping\n");
  yield();
}
```

Pong Thread

```
while (1) {
  printf("pong\n");
  yield();
}
```

yield()

- How does `yield()` work?
- The semantics of yield are that it gives up the CPU to another thread
  - In other words, it context switches to another thread
- So what does it mean for yield to return?
  - It means that *another thread* called yield!
- Execution trace of ping/pong
  - `printf(“ping\n”);`
  - `yield();`
  - `printf(“pong\n”);`
  - `yield();`
  - …

Ping Thread

```
while (1) {
  printf(“ping\n”);
  yield();
}
```

Pong Thread

```
while (1) {
  printf(“pong\n”);
  yield();
}
```

Implementing Yield

```
yield() {
   thread_t old_thread = current_thread;
   append_to_queue(ready_queue, old_thread);
   current_thread = get_next_thread();
   context_switch(old_thread, current_thread);
   return;
}
```

As old thread

As new thread

- The magic step is invoking `context_switch()`
- Why do we need to call `append_to_queue()`?

Thread Context Switch

- The context switch routine does all of the magic
  - Saves context of the currently running thread (old_thread)
    - » Push all machine state onto its stack or TCB
  - Restores context of the next thread
    - » Pop all machine state from the next thread's stack or TCB
  - The next thread becomes the current thread
  - Return to caller as new thread
- This is all done in assembly language

Preemptive Scheduling

- Non-preemptive threads must voluntarily give up CPU
  - A long-running thread will take over the machine
  - Only voluntary calls to `yield`, `sleep`, or `exit` cause a context switch
- Preemptive scheduling uses involuntary context switches
  - Use timer interrupts to regain control of the CPU
  - Timer interrupt handler forces current thread to yield
- Preemptive scheduling is the default in operating systems
  - OS cannot rely on threads to cooperate

Process/Thread Separation

- Separating threads and processes makes it easier to support concurrent applications
  - Concurrency does not require creating new processes
- Concurrency (multithreading) can be very useful
  - Improving program structure
  - Handling concurrent events (e.g., Web requests)
  - Writing parallel programs
- Multithreading is useful with many cores or with one core

Processes: Concurrent Web Server

- Using `fork()` to create new processes to handle requests in parallel is overkill
- Recall our forking Web server:

```
while (1) {
   int sock = accept();
   int child_pid = fork();
   if (child_pid == 0) {
          Handle request, close sock, and exit
   } else {
          Continue
   }
}
```

Threads: Concurrent Web Server

- Instead, we can create a new thread for each request

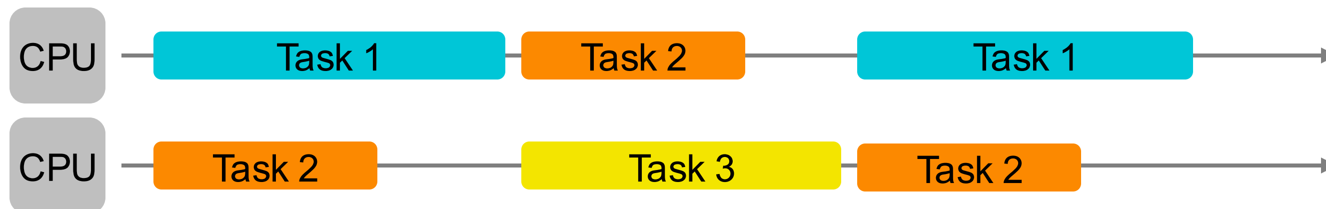
```
while (1) {
   int sock = accept();
   thread_fork(handle_request, sock);
}

handle_request(int sock) {
   Handle client request, close sock, and exit
}
```

Concurrency Questions

- Is it possible to have concurrency with only 1 CPU? Yes
  - Concurrency: multiple tasks in progress at once

- Is parallelism the same thing as concurrency? No
  - Parallelism: multiple tasks running at the same time
- Is this execution concurrent, parallel, or both? Both



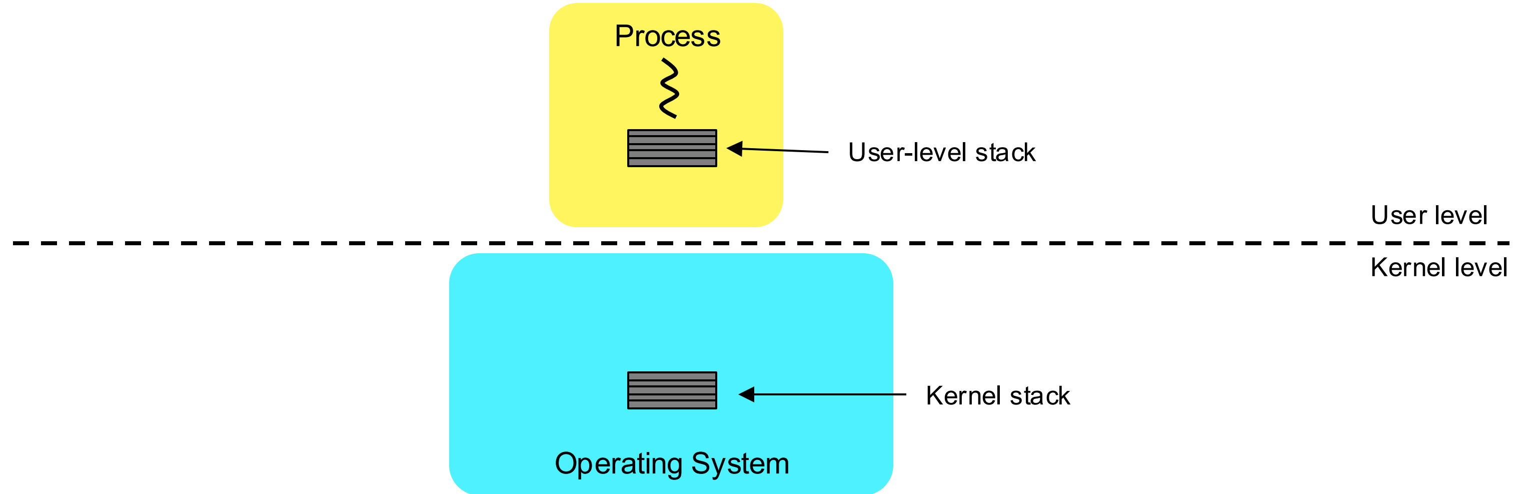
Today's Outline

- Threads
  - What is a thread vs. a process?
- Thread scheduling
  - How should we schedule and execute threads?
- Different kinds of threads
  - Should the OS be aware of threads?

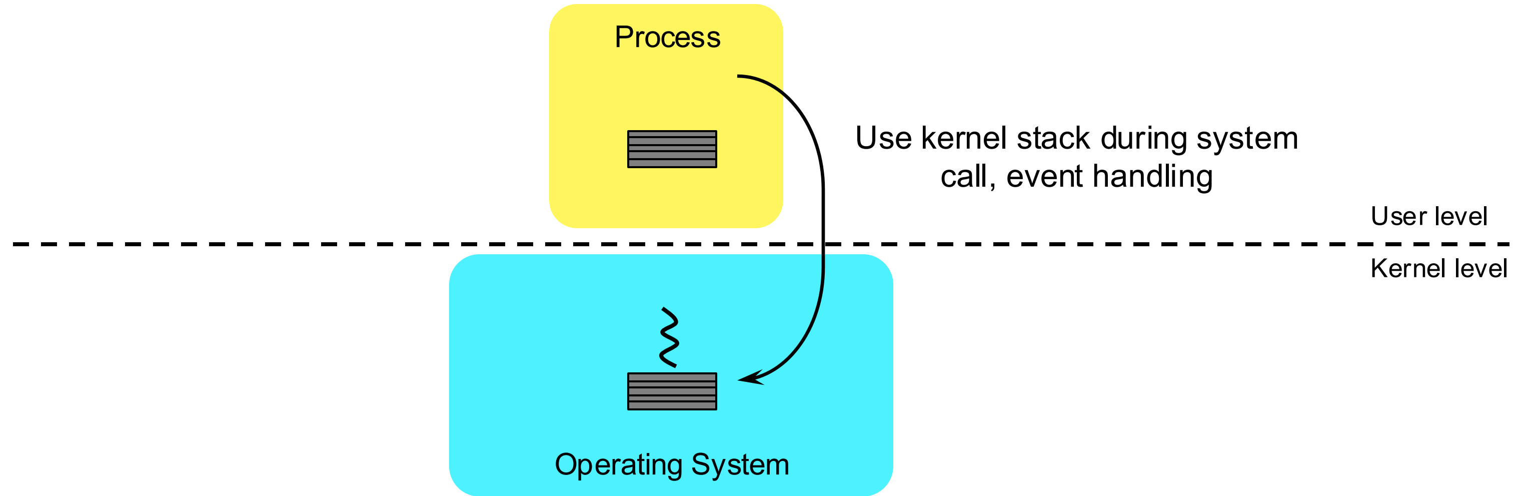
Kernel-Level Threads

- The OS manages threads and processes
  - All thread operations are implemented in the kernel
  - The OS schedules all the threads in the system
- OS-managed threads are called kernel-level threads
  - Windows: threads
  - Solaris: lightweight processes (LWP)
  - POSIX Threads: pthreads

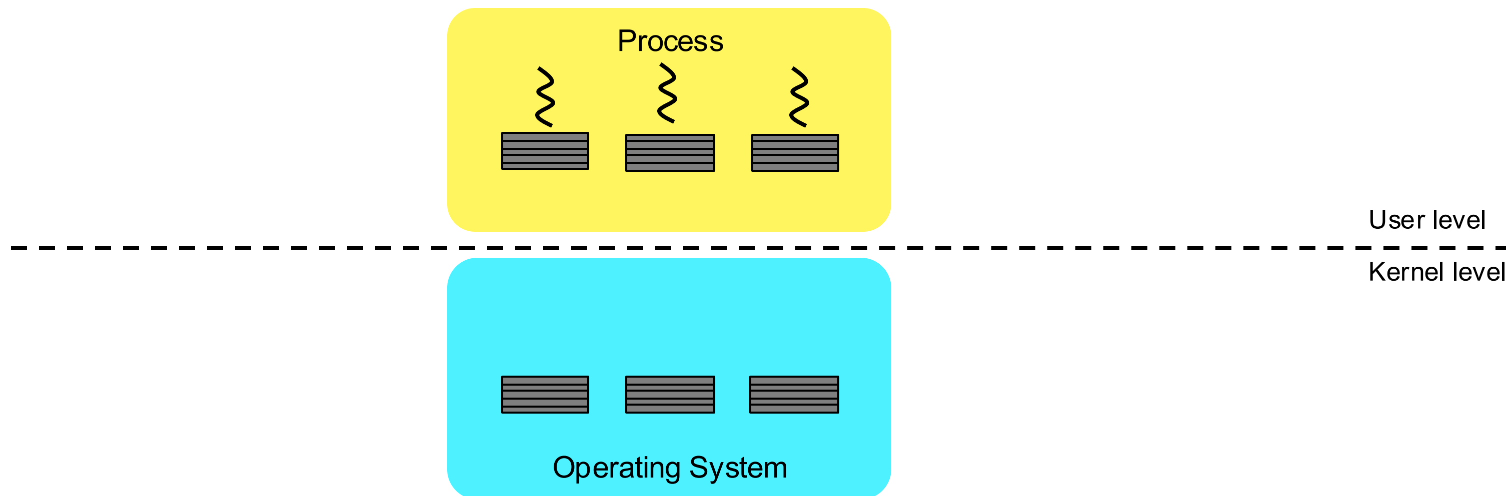
User and Kernel Stacks



Events



Kernel Threads



- Benefits:
  - Multiple kernel threads (OS manages, schedules)
  - Physical parallelism (can run on multiple cores)
  - Multiple separate system calls/events

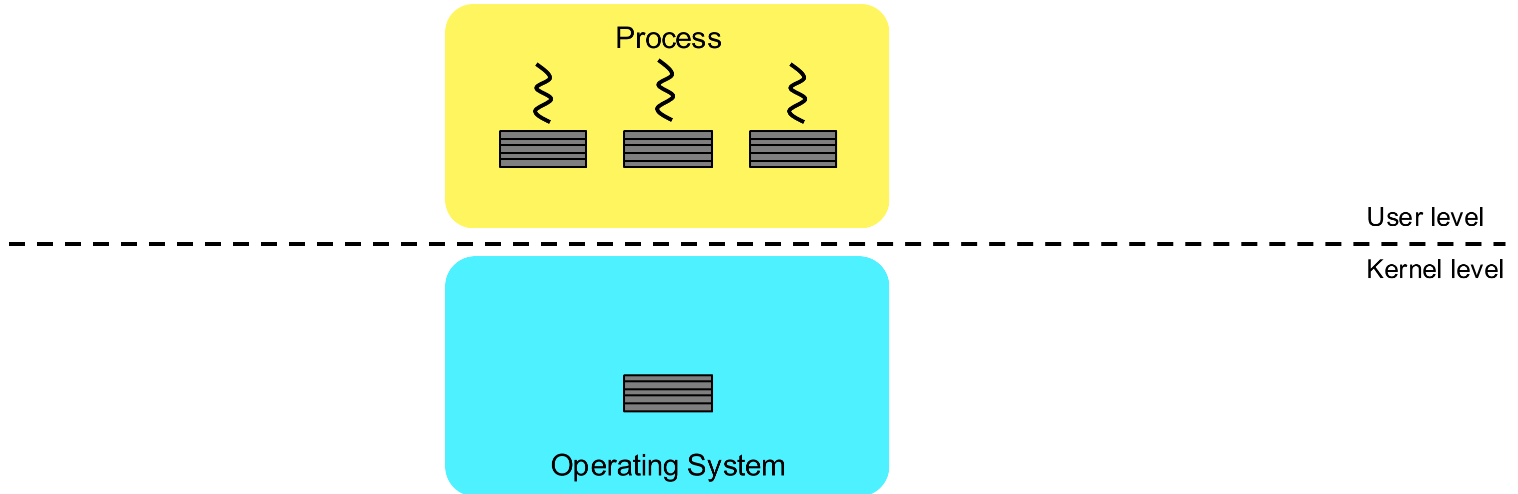
Kernel Thread Limitations

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

User-Level Threads

- What if we hid threads from the kernel?
- To make threads cheap and fast, we can implement them at user level
  - Kernel-level threads: managed by the OS
  - User-level threads: managed entirely by a runtime 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 calls
  - User-level thread operations 10-100x faster than kernel threads

User-Level Threads



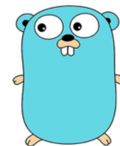
- Multiple user threads multiplexed on one kernel thread
- Downsides:
  - No physical parallelism*
  - Only one system call/event at a time

User-Level Thread Limitations

- User-level threads are not a perfect solution
  - As with everything else, there are tradeoffs
- 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
  - Blocking a process that initiated an I/O, even though the process has other user-level threads that can execute
  - Scheduling a process with no runnable user-level threads

Combining Kernel and User-Level Threads

- Or, use both kernel and user-level threads
  - Multiplex user-level threads on top of kernel-level threads
- Java Virtual Machine (JVM) (also C#, others)
  - Java threads are user-level threads
- Go
  - Go schedules an arbitrary number of goroutines onto an arbitrary number of kernel threads

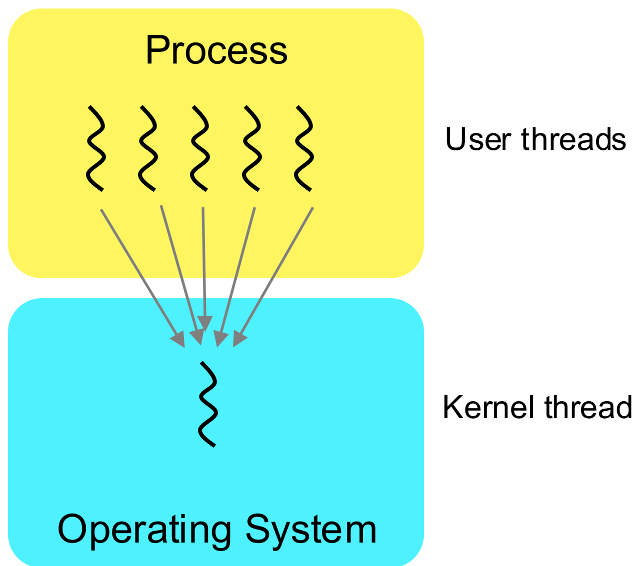


Three Multithreading Models

- Many-to-one
- One-to-one
- Many-to-many

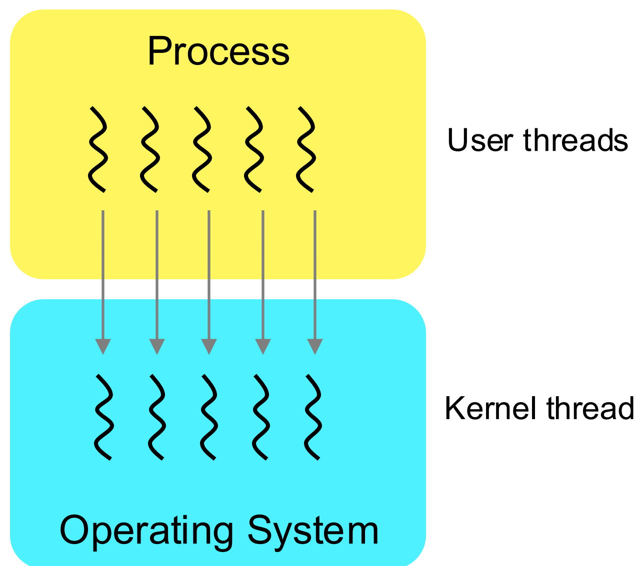
Many-to-One Model

- Many user-level threads mapped to a single kernel thread
- Used in user-level threads



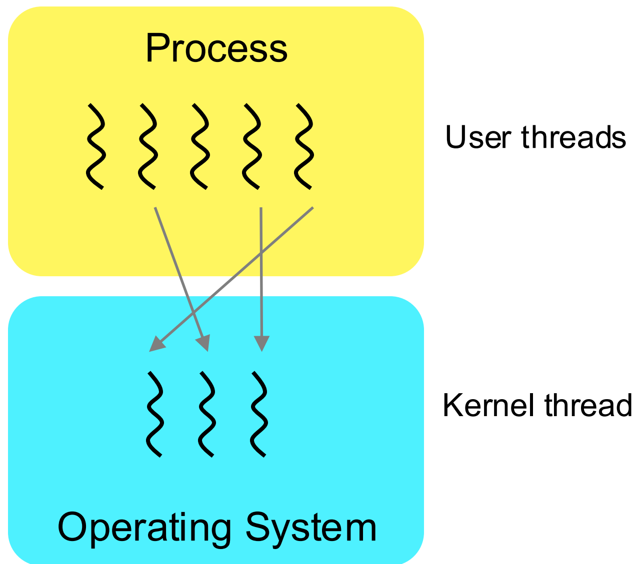
One-to-One Model

- Each user thread maps to a single kernel thread
- Used in kernel-level threads



Many-to-Many Model

- Allows many user-level threads to be mapped to many kernel threads
- Used in user-level threads
- M:N threading model



Processes and Threads Questions

- What abstraction should I use to represent my tasks if…
  - I need to switch very quickly between tasks
    - » User-level threads
  - Each task should not be able to access the files of other tasks
    - » Processes
  - I want to parallelize one application across multiple cores
    - » Kernel-level threads (with or without user-level threads)
  - I want to issue many concurrent requests to the disk
    - » Processes or kernel-level threads

Threading in Nachos

- Nachos emulates a single CPU core (no parallelism)
- Supports both non-preemptive and preemptive scheduling
  - KThread.yield – voluntarily give up the CPU
  - Alarm.timerInterrupt – timer interrupt forces the current thread to yield
- Project 1: implement threads and synchronization within the Nachos kernel
- Project 2: add support for user-level programs

Project 1

- We have released project 1
- Start early!
- For now:
  - Accept your GitHub repo invitation (they expire!)
  - Read the entire project description
  - Work on task 0

Upcoming Tasks

- Discussion tomorrow: project 1 and VS Code
- Read chapters 28-29
- Course Feedback #FinAid
  - On Canvas, due Friday 4/11 11:59 pm
- Homework 1
  - Due Tuesday 4/15 at 11:59 pm
- Project 1
  - Due Tuesday 4/29 at 11:59 pm