CSE 120
Principles of Operating Systems

Spring 2023

Lecture 4: Threads

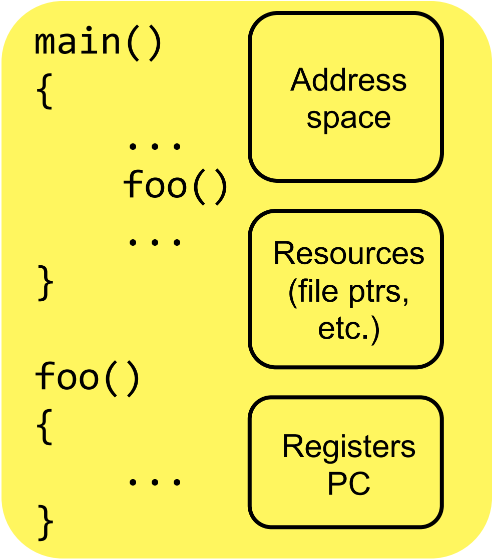
Amy Ousterhout

Administrivia

- Project 0
  - If your grade doesn't make sense, email Kaiyuan
- Homework #1
  - Due 4/18 11:59 pm (next Tuesday), submit via Gradescope
- Project 1
  - Due 4/25
  - Start early!
- Course Feedback #FinAid
  - On Canvas, due Friday 4/14 11:59 pm

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 costly
  - Must create and initialize many data structures
  - Recall 670 LOC for `struct task_struct` in Linux
- Communicating between processes is also costly
  - Processes are supposed to be isolated
  - Communication is mediated by the OS



Process

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

- Applications benefit from executing several tasks in parallel
  - 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
  - Schedule these processes in parallel
- But this is very inefficient
  - Space: PCBs, memory-management state (page tables)
  - Time: create data structures, fork and copy address space

Rethinking Processes

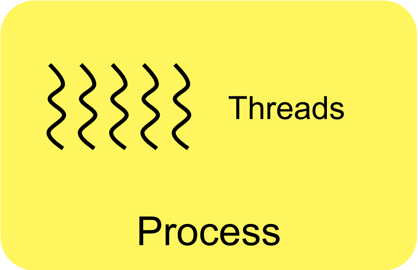
- What is similar in these cooperating processes?
  - They all share the same code and data (address space)
  - They all share the same privileges
  - They all share the same resources (files, sockets, etc.)
- What don't they share?
  - Each has its own execution state: PC, SP, registers
- Key idea: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

Today’s Outline

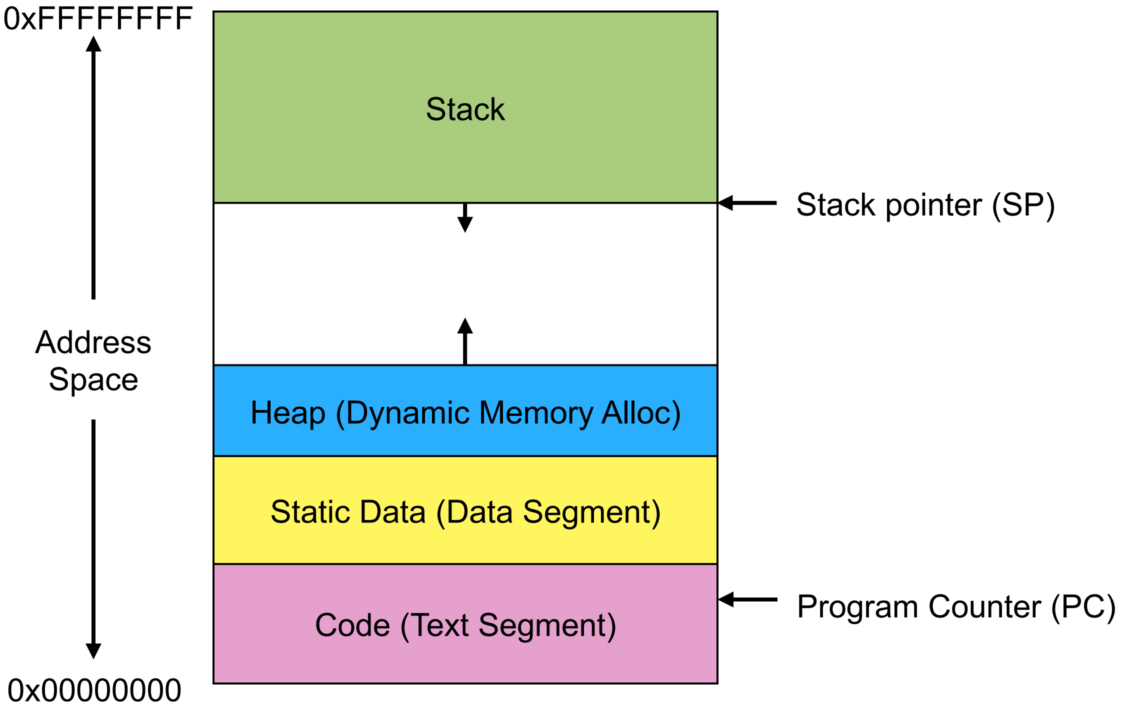
- Threads
  - What is a thread vs. a process?
- Interactions with the OS
  - Should the OS be aware of threads?
- Thread scheduling
  - How should we schedule threads?

Threads

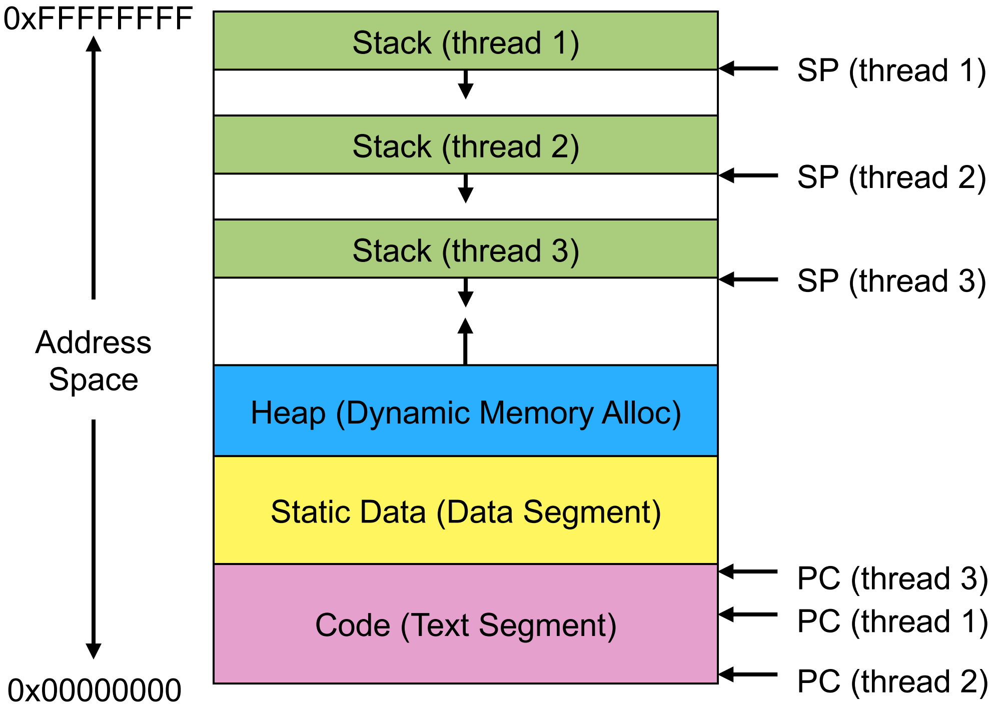
- Modern OSes (Windows, Unix, OS X) separate the concepts of processes and threads
  - Process: address space and resources
  - Thread: a sequential execution stream within a process (PC, SP, registers)
- Each thread is bound to a single process
  - But a process can have multiple threads
- Threads become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



Basic Process Address Space



Basic Process Address Space



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 even useful on a uniprocessor
  - Although today even cell phones are multicore

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

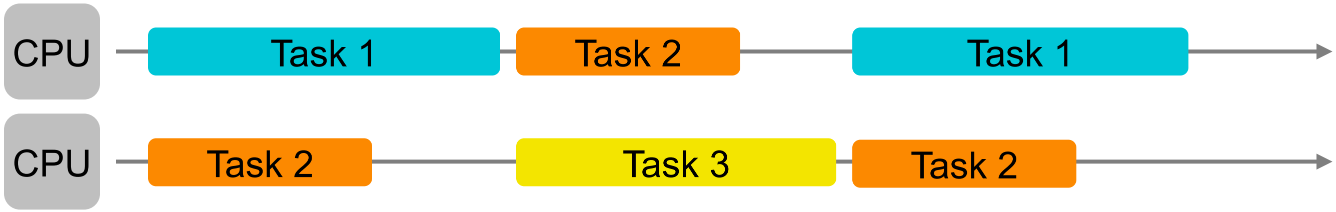
```
while (1) {
   int sock = accept();
   thread_fork(handle_request, sock);
}

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 concurrent, parallel, or both? Both



Today's Outline

- Threads
  - What is a thread vs. a process?
- Interactions with the OS
  - Should the OS be aware of threads?
- Thread scheduling
  - How should we schedule threads?

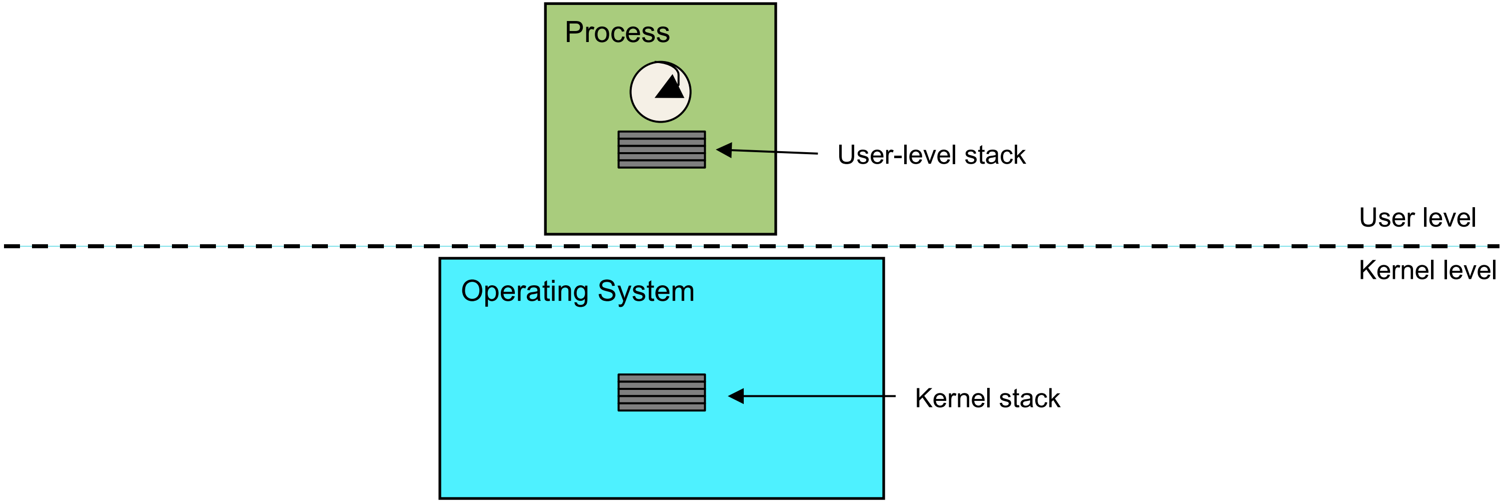
Kernel-Level Threads

- Have the OS manage threads
- Process Control Block
  - Shared information
    - Memory: code/data segments, page tables, stats
    - I/O and files: open file descriptors
  - Per-thread information (Thread Control Block)
    - State (ready, running, or waiting)
    - PC, registers
    - Execution stack

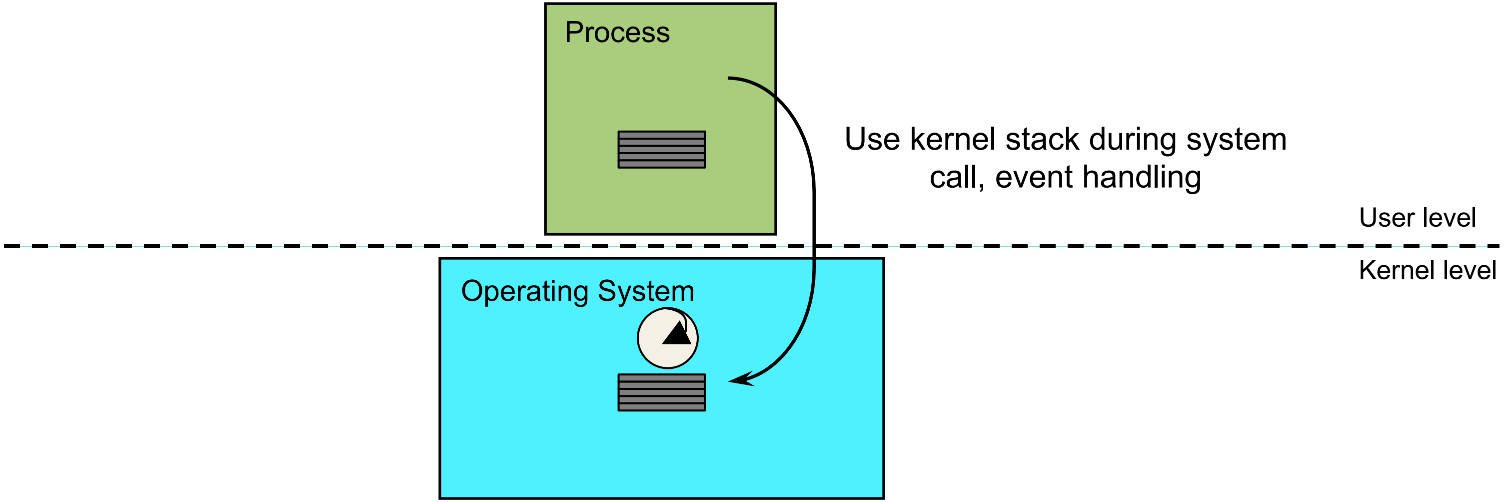
Kernel-Level Threads

- The OS now 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 or lightweight processes
  - Windows: threads
  - Solaris: lightweight processes (LWP)
  - POSIX Threads: pthreads (PTHREAD_SCOPE_SYSTEM)

User and Kernel Stacks

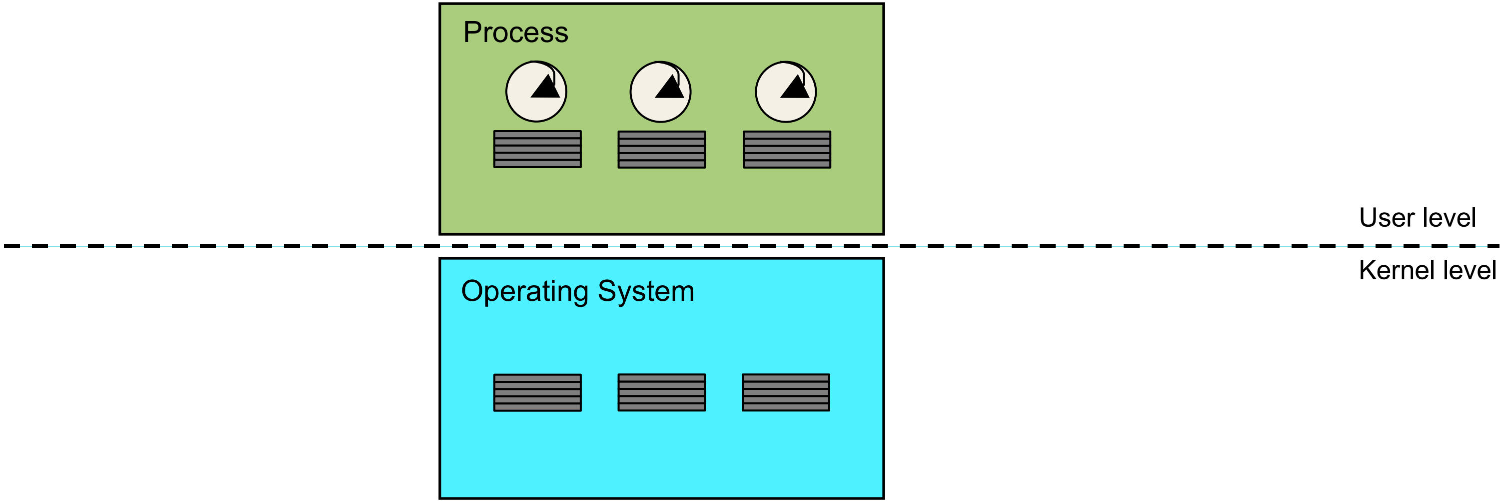


Events



Process
Use kernel stack during system call, event handling
User level
Kernel level
Operating System

Kernel Threads



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

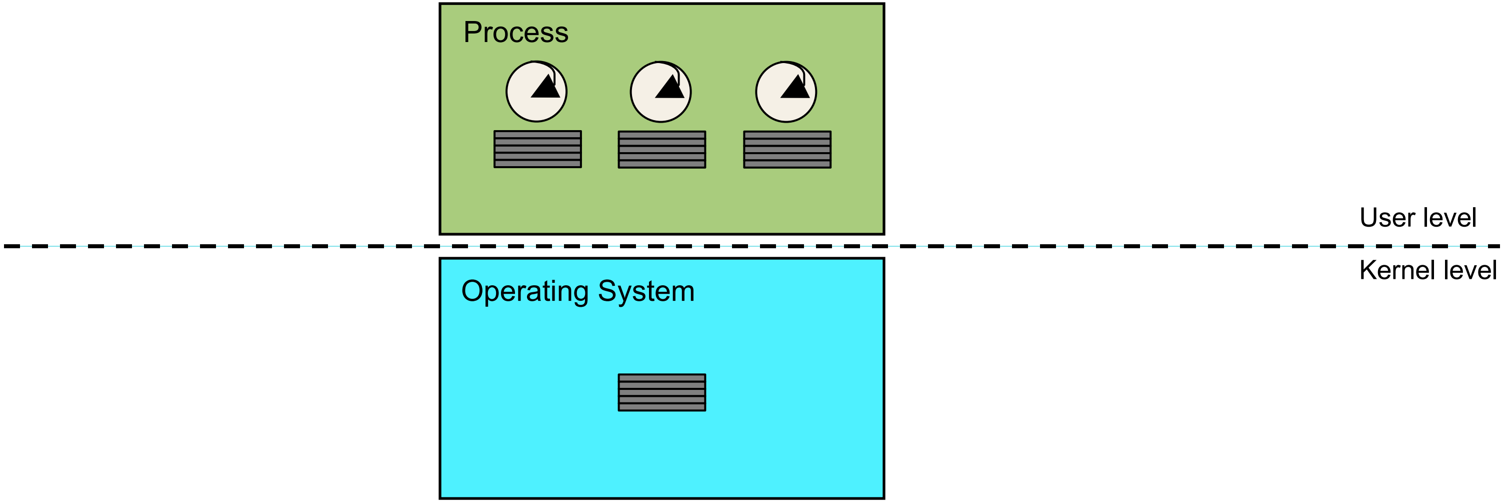
Kernel Thread Limitations

- 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
    - 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 such fine-grained concurrency, want even “cheaper” threads

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 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
  - User-level thread operations 10-100x faster than kernel threads

User-Level Threads



- Multiple user threads (app manages, schedules)
- Multiplexed on one “kernel” thread (no OS support needed)
- Only one system call/event at a time, no physical parallelism*

User-Level Thread Limitations

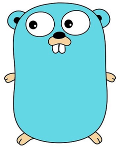
- User-level threads are not a perfect solution
  - As with everything else, there are tradeoffs
- User-level threads are invisible to the OS
  - They are not well integrated with the OS
- As a result, the OS can make poor decisions
  - Blocking a process whose thread initiated an I/O, even though the process has other threads that can execute
  - Scheduling a process with idle threads
  - Unscheduling a process with a thread holding a lock

Kernel vs. User-Level Threads

- Kernel-level threads
  - Integrated with OS (informed scheduling)
  - Slower to create, manipulate, synchronize
- User-level threads
  - Faster to create, manipulate, synchronize
  - Not integrated with OS (uninformed scheduling)
- Understanding the differences between kernel and user-level threads is important
  - Correctness, performance

Combining Kernel and User-Level Threads

- Or, use both kernel and user-level threads
  - Can associate a user-level thread with a kernel-level thread
  - Or, multiplex user-level threads on top of kernel-level threads
- Java Virtual Machine (JVM) (also C#, others)
  - Java threads are user-level threads
  - On modern OSes
    - Can multiplex Java threads on multiple kernel threads
    - Can have more Java threads than kernel threads
- Go
  - Go schedules an arbitrary number of *goroutines* onto an arbitrary number of kernel threads

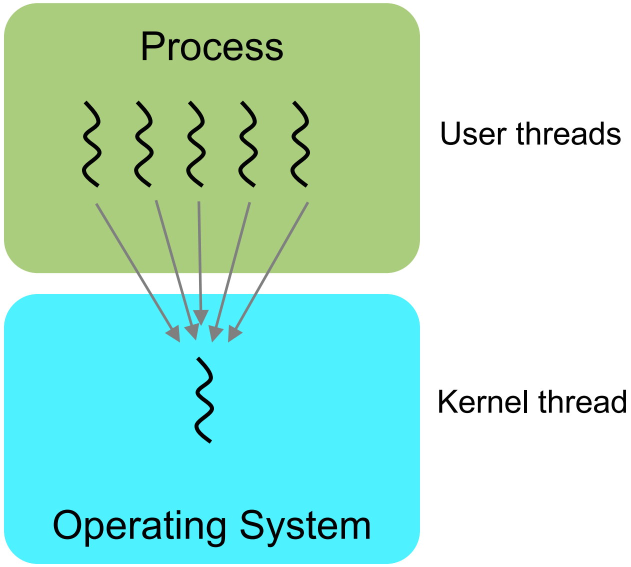


Three Multithreading Models

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

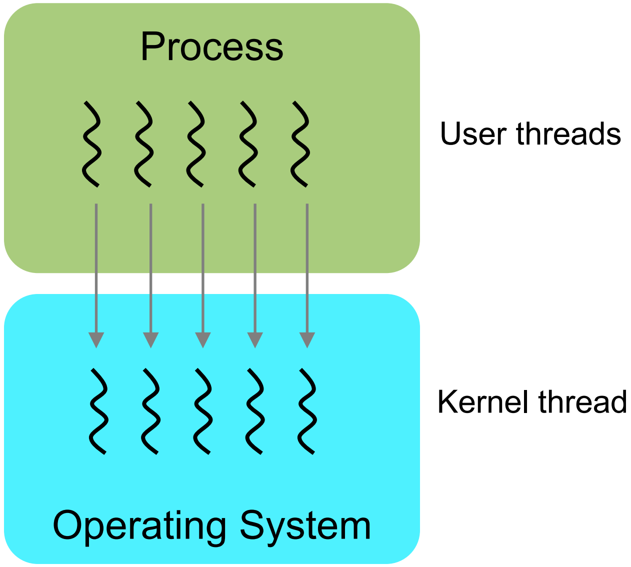
Many-to-One Model

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



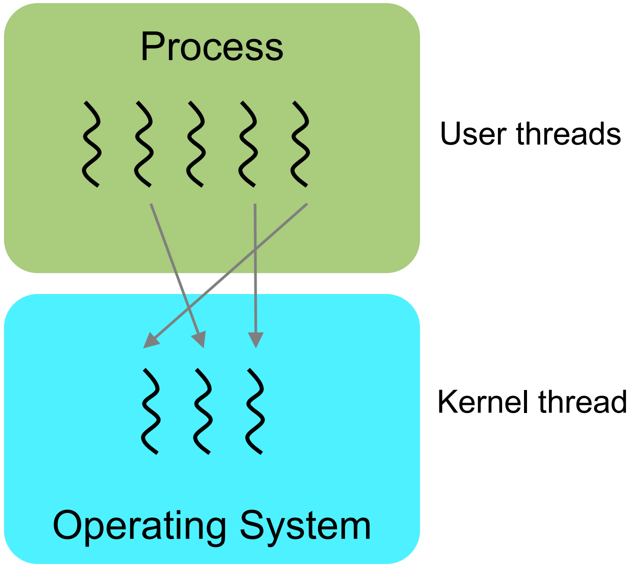
One-to-One Model

- 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



Today's Outline

- Threads
  - What is a thread vs. a process?
- Interactions with the OS
  - Should the OS be aware of threads?
- Thread scheduling
  - How should we schedule threads?

Implementing Threads

- Implementing threads has several aspects
  - Interface
  - Context switch
  - Preemptive vs. non-preemptive scheduling
  - Synchronization (next lecture)
- Focus on user-level threads
  - Kernel-level threads are similar to original process management and implementation in the OS
  - What you will be dealing with in Nachos

Nachos Thread API

- KThread.fork - run a new thread (also "create" in other thread packages)
- KThread.sleep - stop the calling thread (also "stop", "block", "suspend")
- KThread.ready - start the given thread (also "start", "resume")
- KThread.yield - voluntarily give up the processor
- KThread.join - block until another thread finishes (Project 1)
- KThread.finish - terminate the calling thread (also "exit", "destroy")

User-Level Thread Scheduling

- The thread scheduler determines when a thread runs
- It uses queues to keep track of what threads are doing
  - Just like the OS and processes
  - But it is implemented at user-level in a library
- Run queue: threads currently running
- Ready queue: threads ready to run
- Wait queues:
  - How might you implement sleep(time)?
  - Synchronization

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;
   current_thread = get_next_thread();
   append_to_queue(ready_queue, old_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
  - Restores context of the next thread
    - Pop all machine state from the next thread's stack
  - The next thread becomes the current thread
  - Return to caller as new thread
- This is all done in assembly language
  - It works at the level of the procedure calling convention, so it cannot be implemented using procedure calls

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 finish cause a context switch
- Preemptive scheduling uses involuntary context switches
  - Need to regain control of processor asynchronously
  - Use timer interrupt
  - Timer interrupt handler forces current thread to “call” yield
    - See Alarm.timerInterrupt in Nachos

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 only be able to access its own specific set of files
    - » Processes
  - I want to parallelize one application across multiple cores
    - » Kernel-level threads (with or without user-level threads)
  - I want my tasks to work cooperatively, only switching tasks voluntarily
    - » User-level threads
  - I want to issue many concurrent requests to the disk
    - » Threads, either user-level or kernel-level (processes also work, with higher overhead)

Threads Summary

- Threads
  - Threads decouple execution from process management
- Interactions with the OS
  - Kernel-level threads vs. user-level threads
- Thread scheduling
  - Preemptive vs. non-preemptive
- How can our threads cooperate correctly?
  - Next week!

For next class…

- Read chapters 28-29
- HW1 due