Today

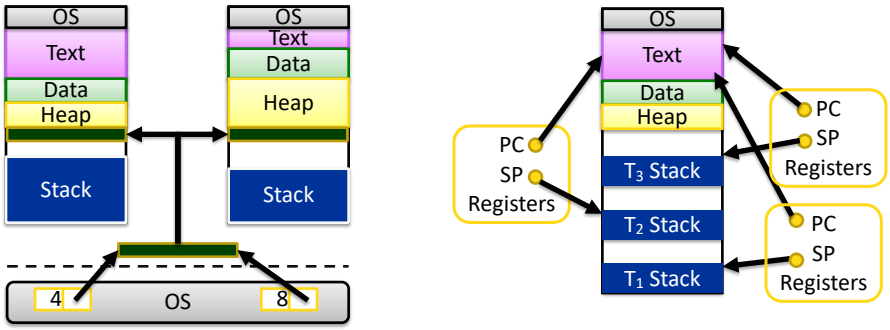
- Threads
  - Implementation models
  - Java API

Review

- What is a thread?
  - What does it contain?
  - How is it related to a process?
  - How is calling `fork` different from and similar to using threads?
- What does threading allow?
- Why do we prefer threads over shared memory?

Shared Memory vs Threads

- These models are equally powerful (for modern thread libraries)



Why do we prefer threads over shared memory?

Threads vs. Interprocess Shared Memory

- Threads: shared virtual address space → LOW context switch overhead

Why threads are called "lightweight"

- Threads: implicit sharing, no extra calls necessary (opening FDs)
- Multiple processes: more protection
  - ONLY explicitly shared memory is accessible to multiple processes
  - Threads can, for example, overwrite each other's stacks

Multithreading vs Alternatives

- Anything that can be done with a multithreaded program can also be done
  - With a single-threaded program
  - With cooperating processes and IPC

How will the multithreaded version compare to these alternatives?

Multithreading Efficiency

- Compared to a single-threaded version of the same program, a multithreaded version may exhibit
  - better responsiveness
  - improved performance
- Compared to an implementation using cooperating processes, a multithreaded implementation will be
  - *more economical* in terms of system resource usage
  - *more efficient* in terms of execution speed
    - Creation
    - Context Switching
    - Communication

Multicore Programming

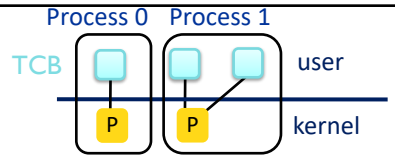
- Types of parallelism
  - Data parallelism – distributes subsets of the same data across multiple cores, same operation on each
  - Task parallelism – distributing threads across cores, each thread performing unique operation
  - Usually implement hybrid of these
- As # of threads grows, so does architectural support for threading

THREAD TYPES

Thread Abstraction vs. Implementation

- Abstraction: multiple execution contexts in a shared VAS
- Implementation decisions:
  - How much should the OS know about threads?
  - How much should the user space process manage about threads?

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 kernel moderating a context switch

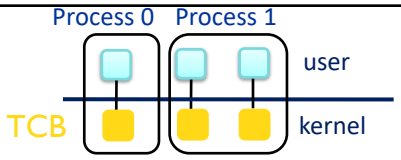
User-Level Threads Life Cycle

Similar to processes and kernel-level threads:

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

What happens if the thread blocks?

Kernel-Level Threads



- A *kernel-level thread* or *kernel thread* is a thread that the OS knows about
  - Every process has at least one kernel-level thread
- Kernel manages and schedules threads
  - And each process has as at least one thread
  - 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 Life Cycle

Similar to processes:

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

Kernel-Level vs User-Level Threads

Kernel-Level Threads

- ✓ System calls do not block the process
- ✓ Switching between threads within the same process is less expensive
  - registers, PC, and SP are changed, memory management info does not
- ✓ Only one scheduler
- – Can be difficult to make efficient

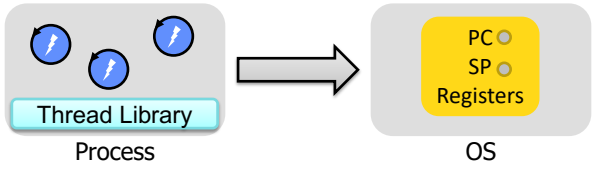
User-level Threads

- ✓ Even faster to create and switch (no system calls or context switches necessary)
  - may be an order of magnitude faster
- ✓ Customizable scheduler
- – All user-level threads in a process block on system calls
  - can use non-blocking versions, if they exist
- – User-level scheduler can conflict with kernel-level scheduler
  - – OS may run a process with only idle threads!

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

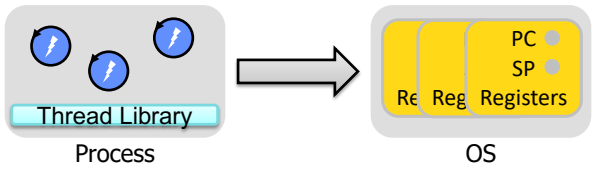
Implementation Option 1 (N:1)

- OS knows *nothing* about threads.
- All execution context stored in *process* memory.
- Threading implemented as a userspace library. Userspace code decides which thread to execute at any given time.



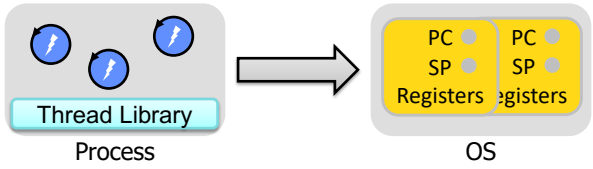
Implementation Option 2 (1:1)

- OS fully aware of threads
- All execution context stored by kernel
- OS creates a *kernel thread* for every new thread and schedules all threads



Implementation Option 3 (N:M)

- OS supports multiple execution contexts, but a process may have more threads than kernel execution contexts.
- Userspace code tracks threads and manages mapping them to kernel execution context.

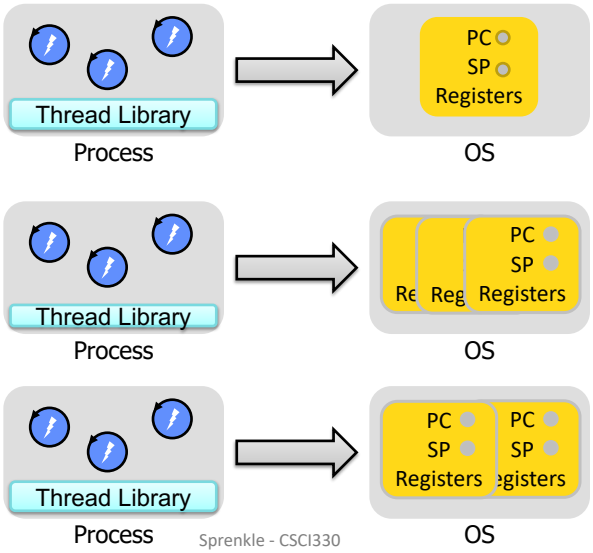


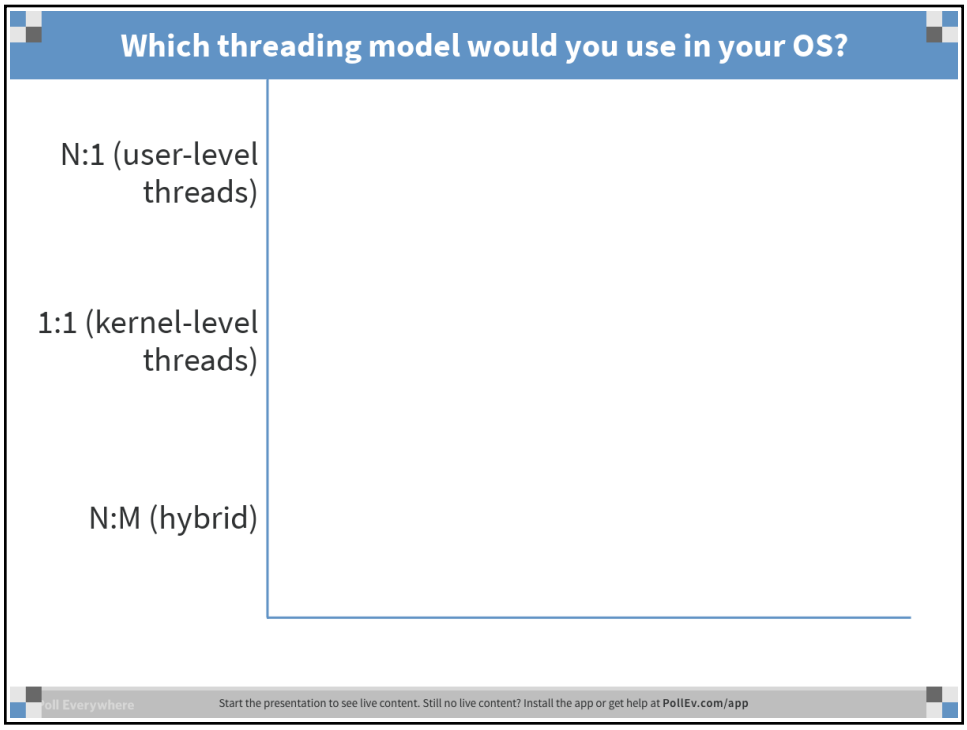
Which threading model would you use in your OS?

A: (N:1)

B: (1:1)

C: (N:M)





Why Use Kernel Threads (1:1)?

- 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
- Simplicity of 1:1 — OS CPU scheduler will do all the scheduling
  - 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

Extending OS Process Model

- Unless we say otherwise, assume 1:1 kernel thread model.
  - OS is aware of every thread and schedules them all independently
  - Thread == "execution context"
- Before: storage for (one) execution context in PCB
  - Registers, PC, SP, kernel stack, etc.
- Now, with threads: PCB contains a collection of threads
  - Each thread represents an execution context

Thread Library

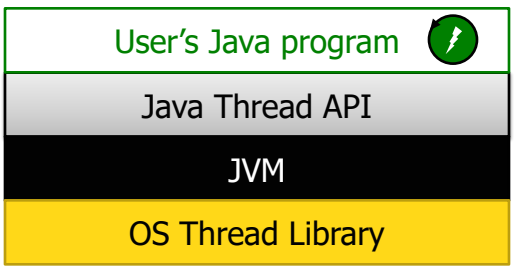
JAVA THREADS

Thread Libraries

- Provides the programmer with an API for creating and managing threads
- Either all in user space with no kernel support
  - invoking a function in the library results in a local function call in user space and not a system call
- Or, implement a kernel-level library supported directly by the operating system
  - code and data structures for the library exist in kernel space
  - Invoking a function in the API for the library typically results in a system call to the kernel.

Java Thread API

- Allows threads to be created and managed directly in Java programs
  - JVM runs on top of a host operating system
  - generally implemented using a thread library available on the host system



Automatically runs in a thread

Java Threads: The Basics

- Extend the Java Thread class

```
class MyThread extends Thread {
    public void run() {
        // do task: your code here
    }
}

…

Thread t1 = new MyThread();
t1.start();
```

Thread Methods

Name	Purpose
`start`	Causes this thread to begin execution; the Java Virtual Machine calls the `run` method of this thread.
`yield()`	A hint to the scheduler that the current thread is willing to yield its current use of a processor.
`join(long millis)`	Waits at most `millis` milliseconds for this thread to die.

Among others….

Java Threads: The Basics

```
public class RunnableTask implements Runnable {

    public RunnableTask(…) {
        // save any arguments or input for the task
(optional)
    }

    @Override
    public void run() {
        // required to implement for Runnable interface
        …
    }

}
…

RunnableTask task = new RunnableTask();
Thread t1 = new Thread(task, "thread1");
t1.start();
```

Java review:
Tradeoffs of extending vs implementing

Example: Jabber

```
class Jabber implements Runnable {
      String str;
      public Jabber(String s){ str = s; }
      public void run() {
            while (true) {
                  System.out.print(str);
                  System.out.println();
            }
      }
}
```

```
public class JabberTest {
      public static void main(String[] args) {
            Jabber jabber1 = new Jabber("1");
            Jabber jabber2 = new Jabber("2");
            Thread t1 = new Thread(jabber1);
            Thread t2 = new Thread(jabber2);
            t1.start();
            t2.start();
      }
}
```

JabberTest.java

Looking Ahead

- Project 2 due tonight
- Exam released after class on Wednesday
  - Due Friday at 5 p.m.
  - Bring questions on Wednesday