

# **CSE 120**

# **Principles of Operating Systems**

**Fall 2023**

**Lecture 4: Threads**

Geoffrey M. Voelker

# Processes

---

- Recall that a process includes many things
  - ◆ An address space (defining all the code and data pages)
  - ◆ OS resources (e.g., open files) and accounting information
  - ◆ Execution state (PC, SP, regs, etc.)
- **Creating a new process is costly** because of all of the data structures that must be allocated and initialized
  - ◆ Recall `struct proc` in Solaris
- **Communicating between processes is also costly** because most communication goes through the OS
  - ◆ Overhead of system calls and copying data

# Concurrent Programs

---

- Recall our Web server example that forks off copies of itself to handle multiple simultaneous requests
  - ◆ Or any parallel program that executes on a multiprocessor
- To execute these programs 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
  - ◆ 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.

# Rethinking Processes

---

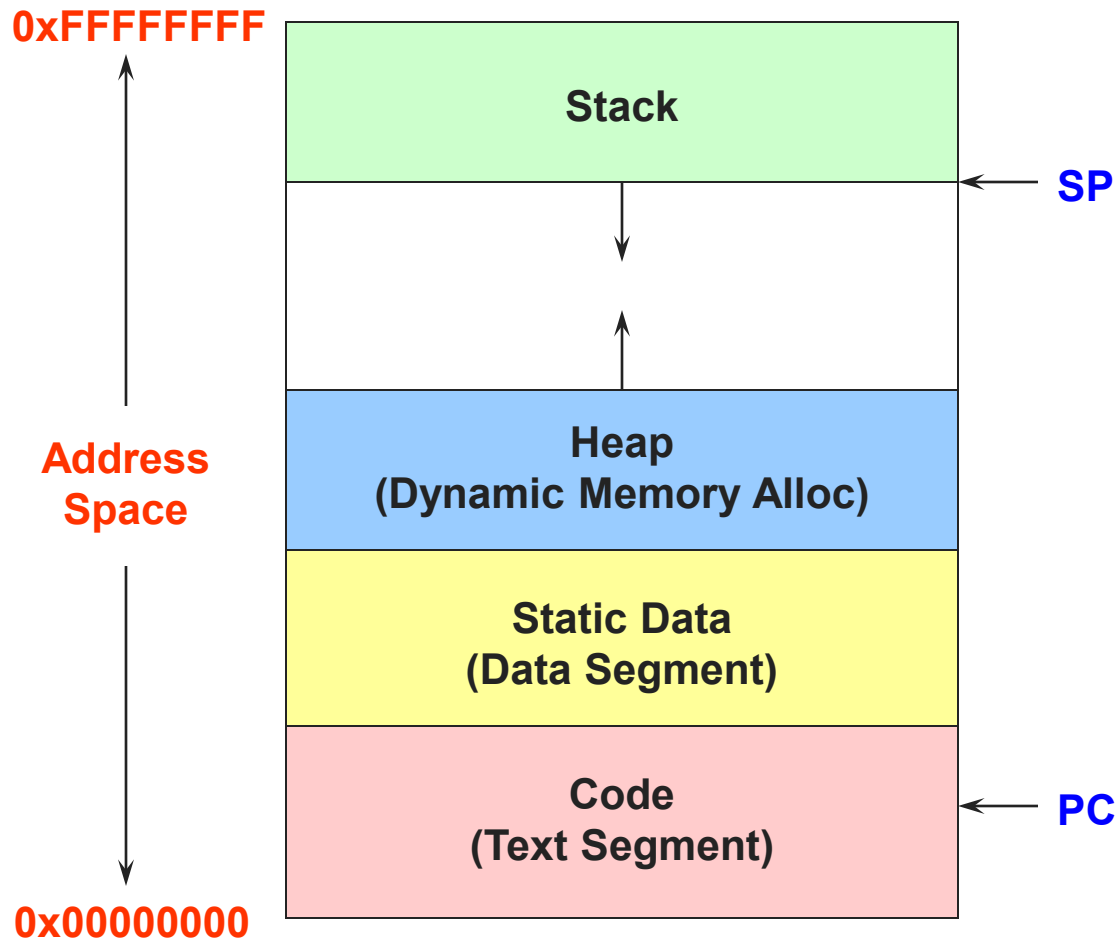
- What is similar in these cooperating processes?
  - ◆ They all share the same code and data (address space)
  - ◆ They all share the same privileges (user ID)
  - ◆ They all share the same resources (files, sockets, etc.)
- What don't they share?
  - ◆ Each has its own execution state: PC, SP, and 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
- Exec state also called **thread of control**, or **thread**

# Threads

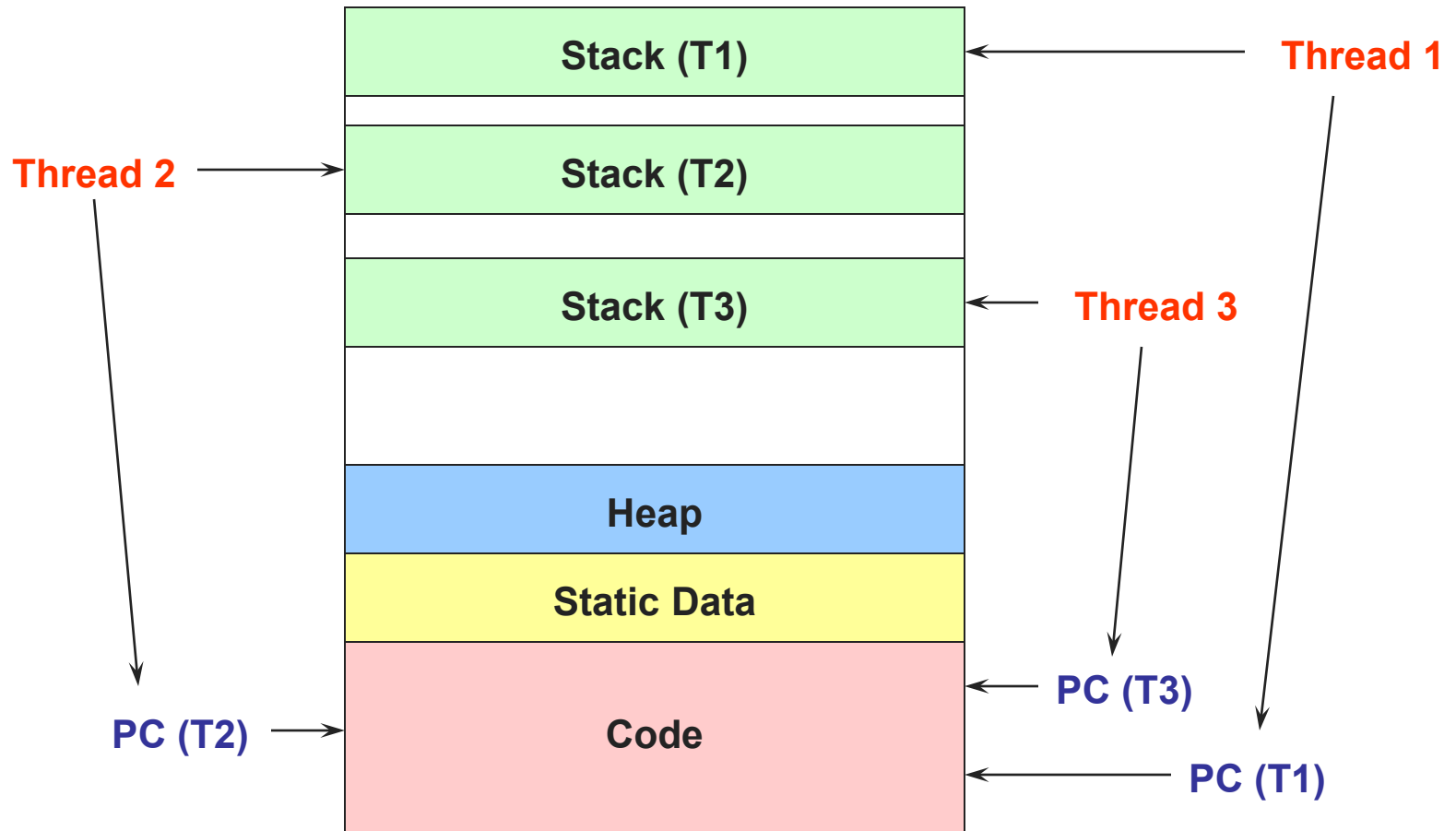
---

- Modern OSes (Windows, Unix, OS X) separate the concepts of processes and threads
  - ◆ The **thread** defines a sequential execution stream within a process (PC, SP, registers)
  - ◆ The **process** defines the address space and general process attributes (everything but threads of execution)
- A thread is bound to a single process
  - ◆ Processes, however, can have multiple threads
- Threads become the basic unit of scheduling
  - ◆ Processes are now the **containers** in which threads execute
  - ◆ Processes become static, threads are the dynamic entities

# Basic Process Address Space



# Threads in a Process



# Process/Thread Separation

---

- Separating threads and processes makes it easier to support multithreaded 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
- So multithreading is even useful on a uniprocessor
  - ◆ Although today even cell phones are multicore

# Process: Concurrent Servers

---

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

```
while (1) {  
    int sock = accept();  
    if ((child_pid = fork()) == 0) {  
        Handle client request  
        Close socket and exit  
    } else {  
        Close socket  
    }  
}
```

# Threads: Concurrent Servers

---

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

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

```
handle_request(int sock) {  
    Process request  
    close(sock);  
}
```

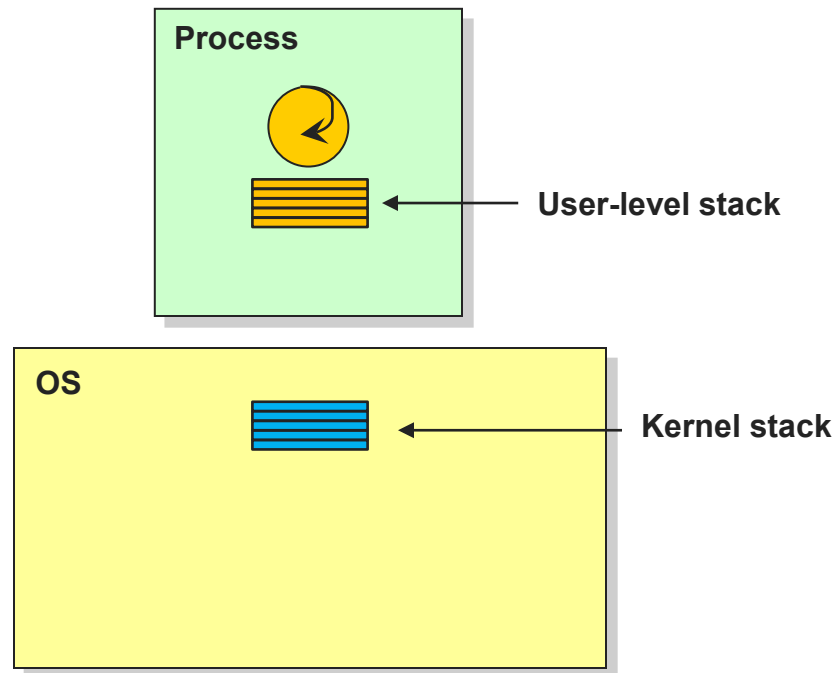
# Kernel-Level Threads

---

- We have taken the execution aspect of a process and separated it out into threads
  - ◆ To make concurrency cheaper
- As such, 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**
  - ◆ Linux (pthreads), Windows (threads), Solaris (lightweight processes (LWPs)), ...

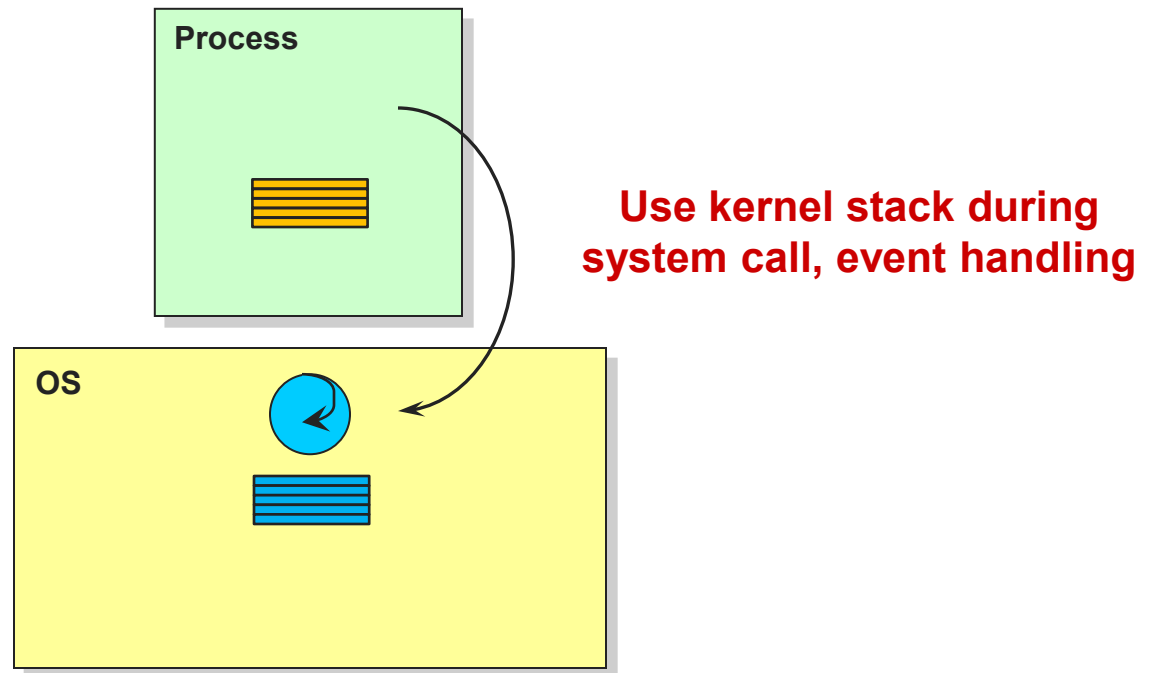
# User and Kernel Stacks

---



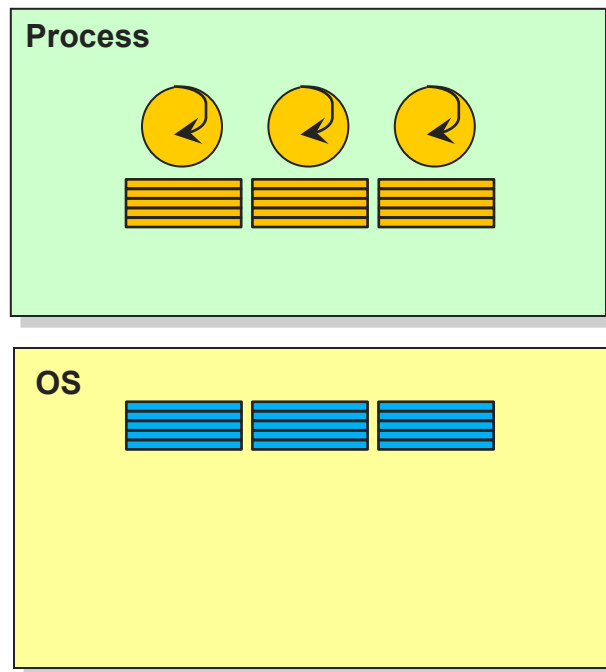
# System Calls / Events

---



# Kernel Threads

---



- Multiple kernel threads (OS manages, schedules)
- Physical parallelism (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, need even “cheaper” threads

# User-Level Threads

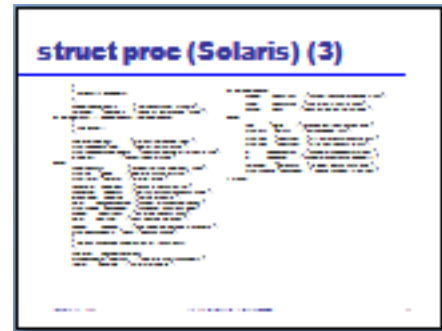
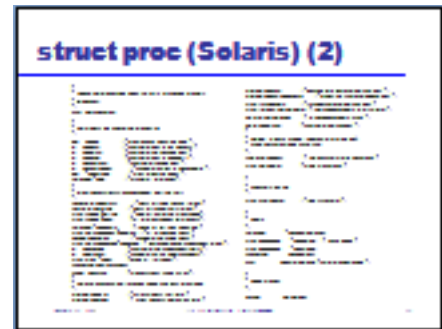
---

- 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**
    - » User-level thread operations **100x faster** than kernel threads
  - ♦ Early JVMs (Thread), Go (goroutines)

# Small and Fast...

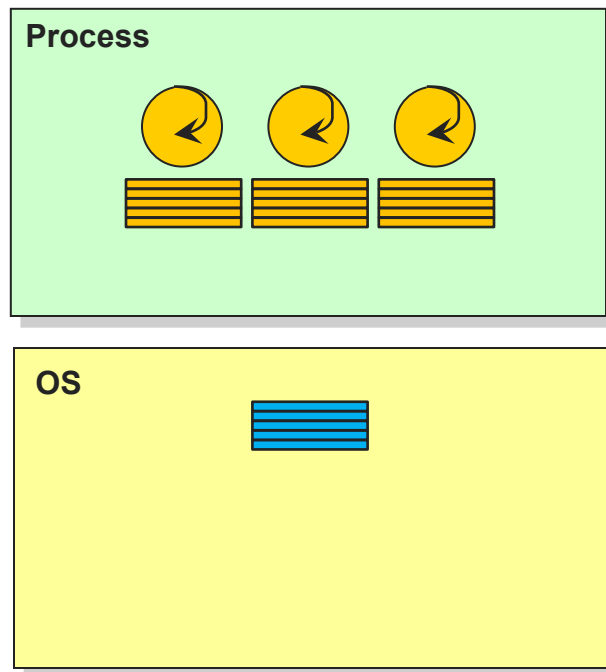
- Nachos thread class

```
public class KThread {  
    int status;  
    String name;  
    Runnable target;  
    TCB tcb;  
    int id;  
    <Methods>  
};
```



# User 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

# U/L 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
  - ◆ Unscheduling a process with a thread holding a lock

# Kernel vs. User Threads

---

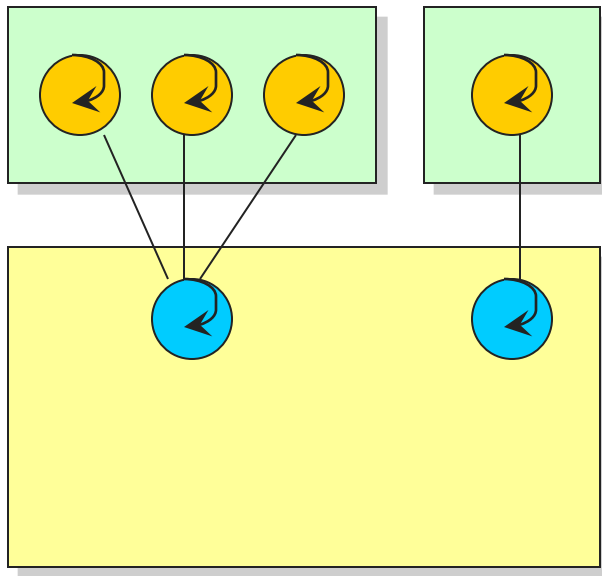
- Kernel-level threads
  - ◆ Integrated with OS (informed scheduling)
  - ◆ Slower to create, manipulate, synchronize
- User-level threads
  - ◆ Faster to create, manipulate, synchronize
  - ◆ Not integrated with OS (uninformed scheduling)

# Kernel and User Threads

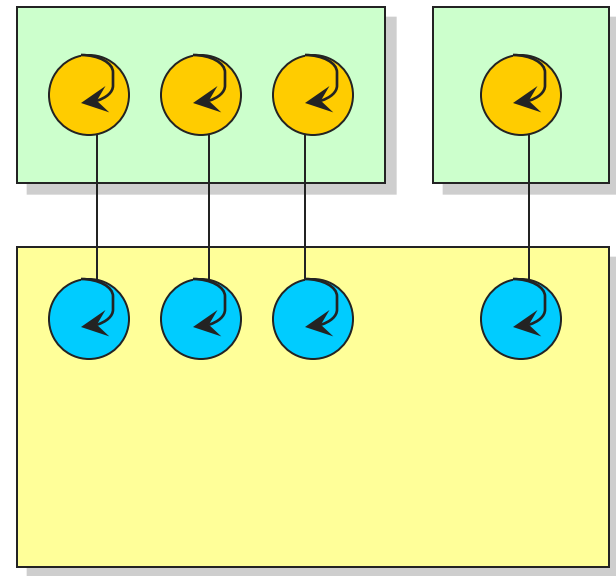
---

- 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)
  - ♦ On older Unix, only one “kernel thread” per process
    - » Multiplex all Java threads on this one kernel thread (M:1)
  - ♦ On modern OSes
    - » Each Java thread uses a separate kernel thread (1:1)
    - » Can have more Java threads than kernel threads (M:N)

# User and Kernel Threads



**Multiplexing user-level threads  
on a single kernel thread for  
each process**



**Multiplexing user-level threads  
on multiple kernel threads for  
each process**

# Implementing Threads

---

- Implementing threads has several aspects
  - ◆ Interface
  - ◆ Context switch
  - ◆ Preemptive vs. non-preemptive scheduling
  - ◆ Synchronization (next lecture)

# Nachos Thread API

---

- **KThread.fork**
  - ◆ Run a new thread (also “create” in other thread packages)
- **KThread.sleep**
  - ◆ Block the calling thread (also “stop”, “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”)

# Thread Scheduling

---

- The thread scheduler determines when a thread runs
- It uses queues to keep track of what threads are doing
- Run queue: Threads currently running
  - ♦ Just one with Nachos
- Ready queue: Threads ready to run
- Wait queues: Threads blocked (asleep) waiting
  - ♦ Synchronization, Alarm

# Non-Preemptive Scheduling

---

- Threads voluntarily give up the CPU with yield

Ping Thread

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

Pong Thread

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

- What is the output of running these two threads?

# yield

---

- Wait a second. 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();`
  - ◆ ...

# 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

# Threads Summary

---

- The operating system as a large multithreaded program
  - ◆ Each process executes as a thread within the OS
- Multithreading is also very useful for applications
  - ◆ Efficient multithreading requires fast primitives
  - ◆ Processes are too heavyweight
- Solution is to separate threads from processes
  - ◆ Kernel-level threads much better, but still significant overhead
  - ◆ User-level threads even better, but not well integrated with OS
- Now, how do we get our threads to correctly cooperate with each other?
  - ◆ Synchronization...

# Next time...

---

- Read Chapters 28, 29
- Homework #1 due