

CSE 120

Principles of Operating Systems

Fall 2007

Lecture 4: Threads

Keith Marzullo

Announcements

- Homework #1 due now

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
 - ◆ ...which does not even include page tables, perhaps TLB flushing, etc.
- Communicating between processes is costly because most communication goes through the OS
 - ◆ Overhead of system calls and copying data

Parallel Programs

- Also 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
 - ◆ Have the OS 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.

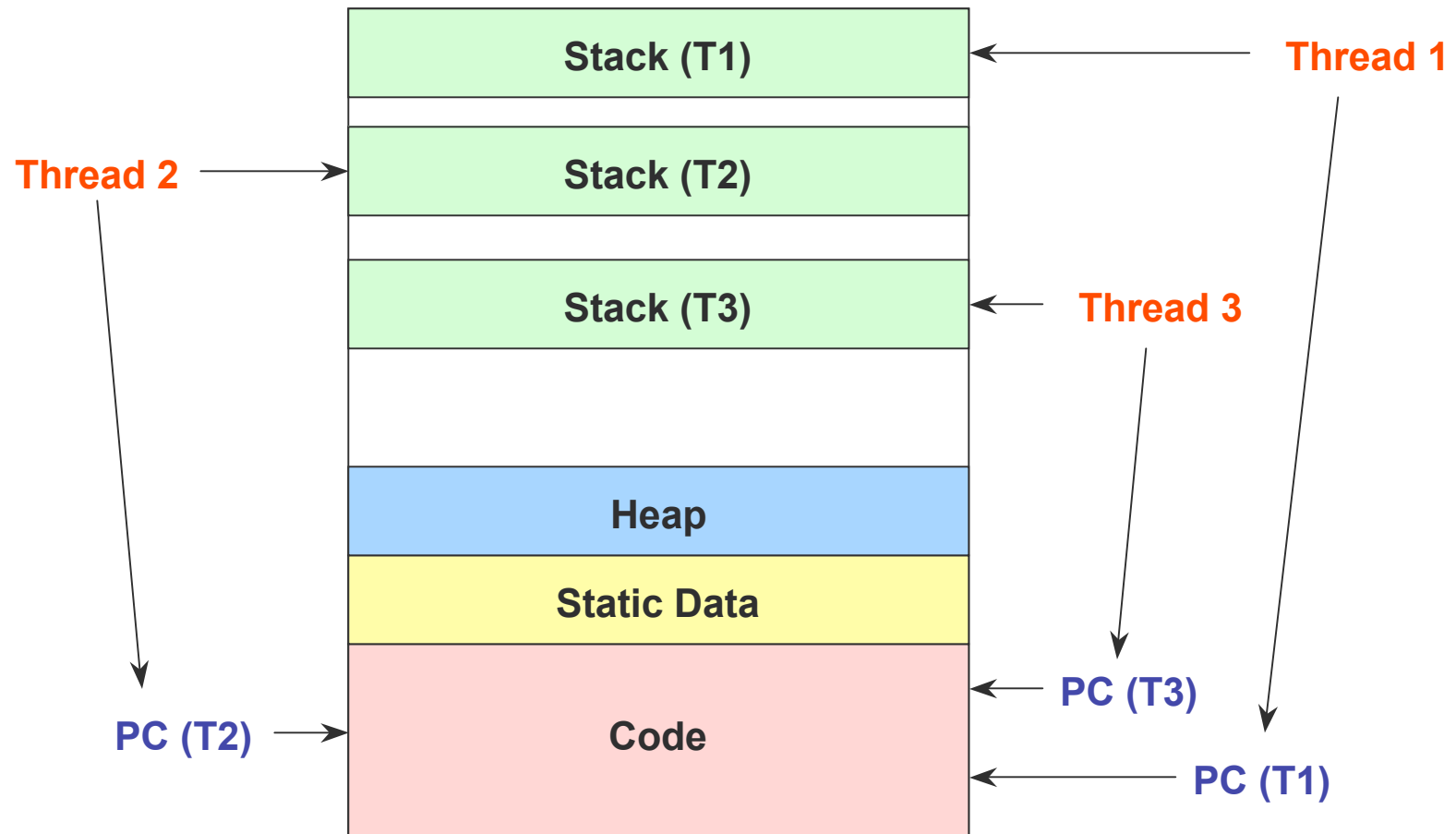
Rethinking Processes

- What is similar in these cooperating processes?
 - ◆ They all share the same code and data (address space)
 - ◆ They all share the same privileges
 - ◆ 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**

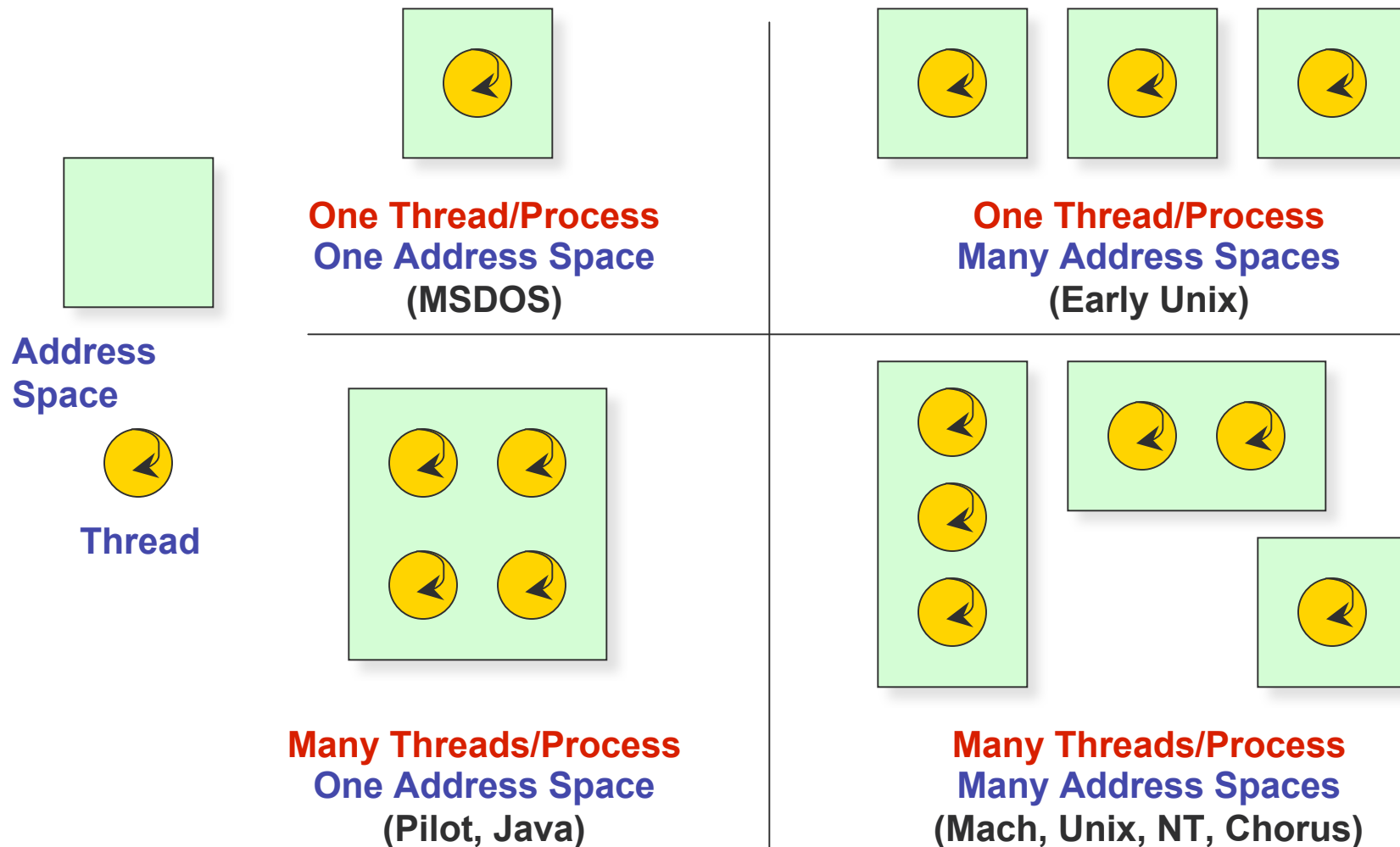
Threads

- Modern OSes (Mach, Chorus, NT, modern Unix) 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 unit of scheduling
 - ◆ Processes are now the **containers** in which threads execute
 - ◆ Processes become static, threads are the dynamic entities

Threads in a Process



Thread Design Space



Process/Thread Separation

- Separating threads and processes makes it easier to support multithreaded applications
 - ◆ Creating 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

Threads: Concurrent Servers

- Using fork() to create new processes to handle requests in parallel is overkill for such a simple task
- 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 of the threads in the system
- OS-managed threads are called **kernel-level threads** or **lightweight processes**
 - ◆ NT: **threads**
 - ◆ Solaris: **lightweight processes (LWP)**

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 too much 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
 - » No kernel involvement
 - ◆ User-level thread operations 100x faster than kernel threads

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
- Solving this requires communication between the kernel and the user-level thread manager

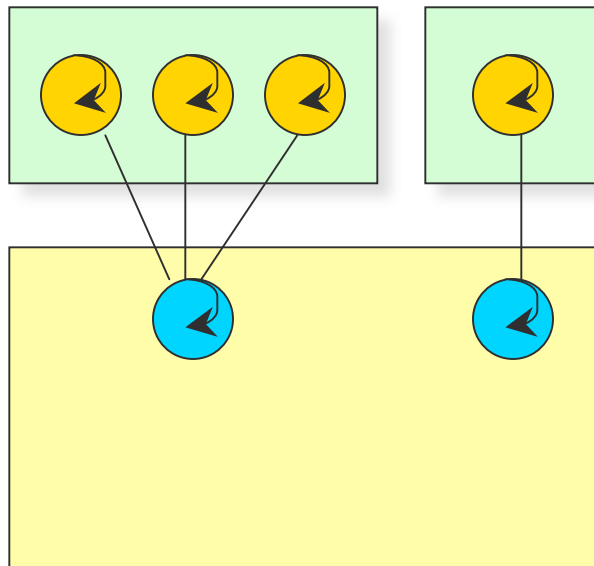
Kernel vs. User Threads

- Kernel-level threads
 - ◆ Integrated with OS (informed scheduling)
 - ◆ Slow to create, manipulate, synchronize
- User-level threads
 - ◆ Fast to create, manipulate, synchronize
 - ◆ Not integrated with OS (uninformed scheduling)
- Understanding the differences between kernel and user-level threads is important
 - ◆ For programming (correctness, performance)
 - ◆ For test-taking

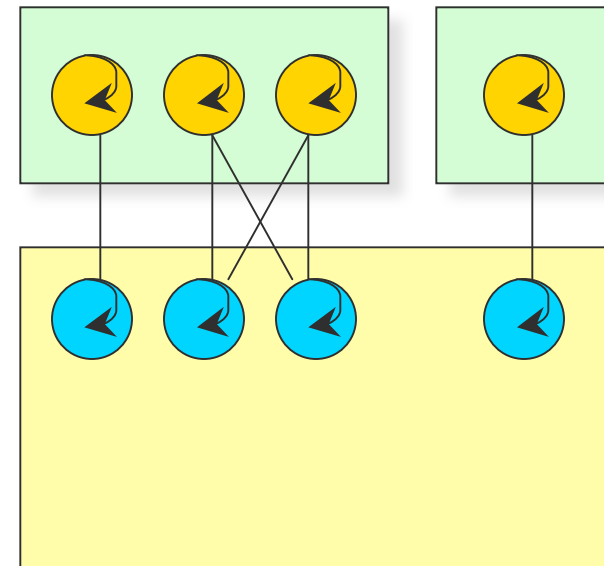
Kernel and User Threads

- Another possibility is to 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)
 - ◆ Java threads are user-level threads
 - ◆ On older Unix, only one “kernel thread” per process
 - » Multiplex all Java threads on this one kernel thread
 - ◆ On NT, modern Unix
 - » Can multiplex Java threads on multiple kernel threads
 - » Can have more Java threads than kernel threads
 - » Why?

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 a number of issues
 - ◆ Interface
 - ◆ Context switching
 - ◆ 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

Thread Project Interface

- `t_fork(ThreadFunc f, any_ptr v)`
 - ◆ Create a new thread of control
 - ◆ thread terminates when `f` returns
 - ◆ Also `t_start(f, v)`
- `t_yield()`
 - ◆ Voluntarily give up the process

Thread Scheduling

- The thread scheduler determines when a thread runs
 - ◆ Project 1: non-preemptive round-robin
 - ◆ Project 2: preemptive priority
- It uses queues to keep track of what threads are doing
 - ◆ Just like the OS and processes
 - ◆ But it is implemented at user-level
- Ready queue: Threads ready to run
 - ◆ Head of queue can be currently running thread.
- Do we need any wait queues?

Non-Preemptive Scheduling

- Threads voluntarily give up the CPU with `t_yield`

Ping Thread

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

Pong Thread

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

- What is the output of running these two threads?

t_yield()

- How does t_yield() work?
- The semantics of t_yield are that it gives up the CPU to another thread
 - ♦ In other words, it **context switches** to another thread
 - ♦ But, in terms of the thread's state, it's a no-op.
- So what does it mean for t_yield to return?
 - ♦ It means that *another thread* called t_yield!
- Execution trace of ping/pong
 - ♦ `printf("ping\n");`
 - ♦ `t_yield();`
 - ♦ `printf("pong\n");`
 - ♦ `t_yield();`
 - ♦ ...

Thread Context Switch

The context switch routine does all of the magic

- Saves context of the currently running thread (`setjmp`)
- Move thread descriptor to the end of the ready queue
- Restores context of the next thread (`longjmp`)
 - ◆ This is the thread at the head of the ready queue
 - ◆ Returns to `setjmp` that saved this thread's context

What will we do in Project 2?

Pre-emptive version of round-robin

- Use a timer interrupt to pre-empt the processor from the currently running process
- Similar to having thread call `t_yield()`

Order ready queue with respect to thread priorities

- When a thread lowers its priority, it may end up being preempted (because it moves from the head of the ready list)

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 Chapter 6.1—6.6