CSCI3150 Introduction to Operating Systems

Lecture 4: Threads

Hong Xu

https://github.com/henryhxu/CSCI3150

Processes

- Recall that a process includes many things
  - An address space (all the code and data)
  - OS resources (e.g., open files) and accounting information
  - Execution state (PC, SP, regs, etc.)
- Creating a new process is costly because of all 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.
- Communication between processes is costly because it goes through the OS
  - Overhead of system calls and copying data

Parallel Programs

- 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
- This situation is very inefficient
  - Space: PCB, page tables, etc.
  - Time: create data structures, fork and copy addr space, etc.
- Is it possible to have more efficient, yet cooperative "processes"?

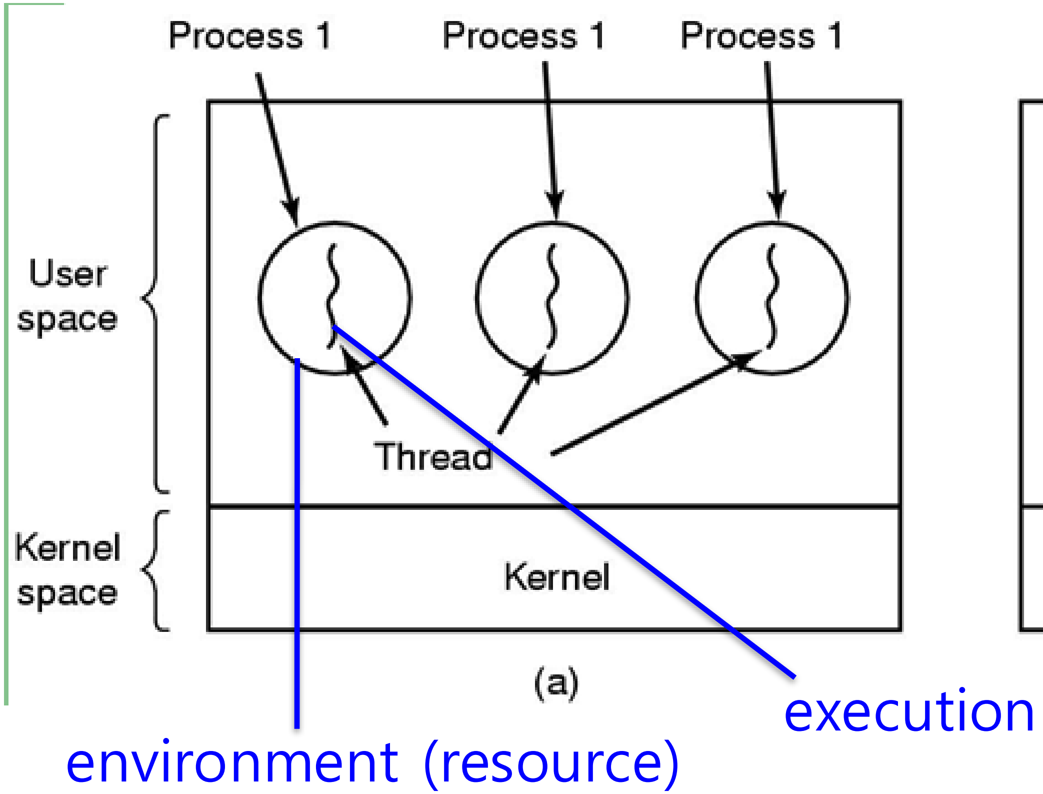
Rethinking Processes

- 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 decouple 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, 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
- 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

Threads: lightweight processes

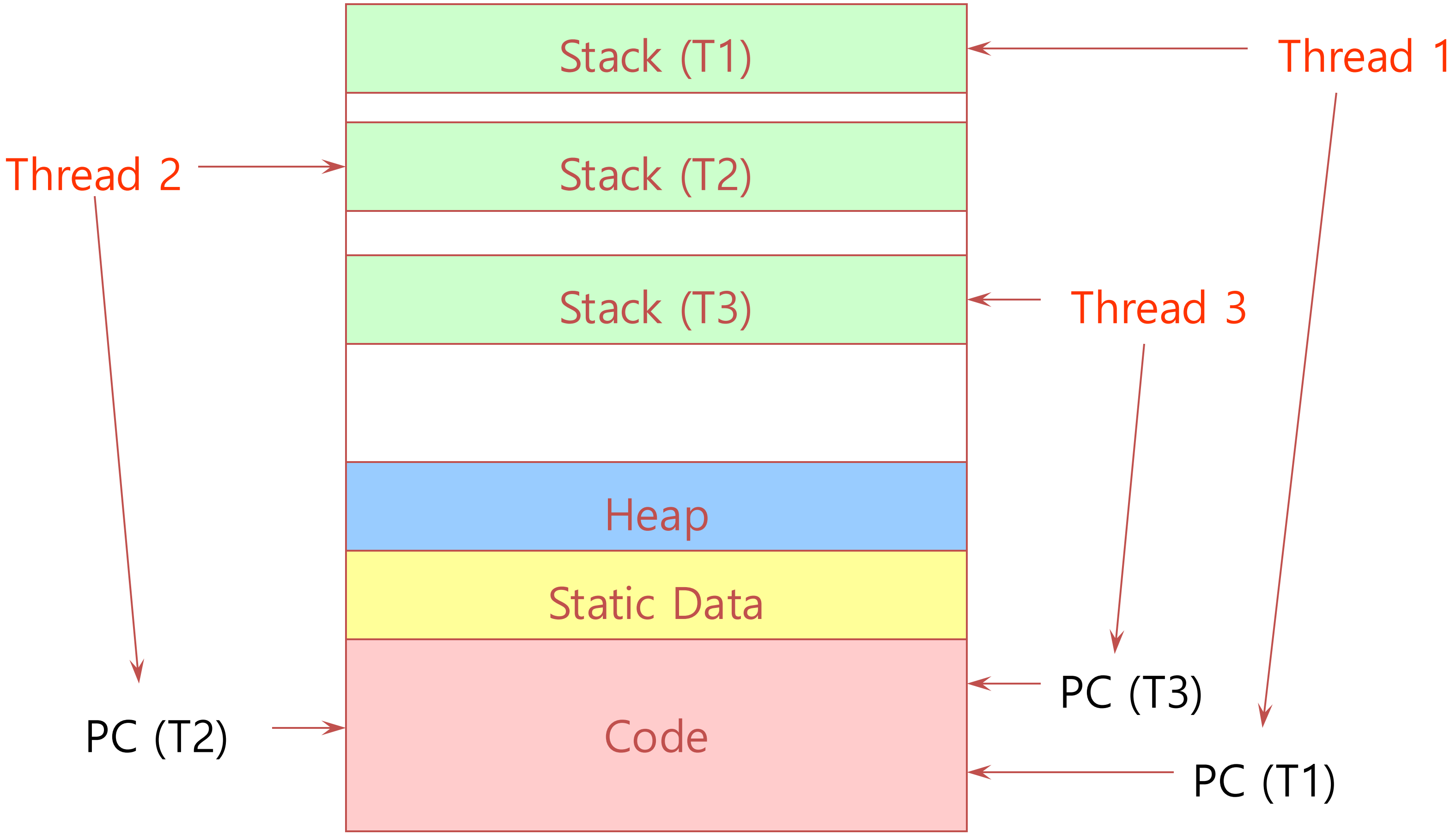


(a) Three processes each with one thread
(b) One process with three threads

The thread model

- Shared information
  - Processor info: parent process, time, etc
  - Memory: segments, page table, and stats, etc
  - I/O and file: communication ports, directories and file descriptors, etc.
- Private state
  - State (ready, running and blocked)
  - Registers
  - Program counter
  - Execution stack
- Each thread executes independently

Threads in a Process



Threads: Concurrent Servers

- Using fork() to create new processes to handle requests in parallel is an overkill for such a simple task
- Recall our forking Web server:

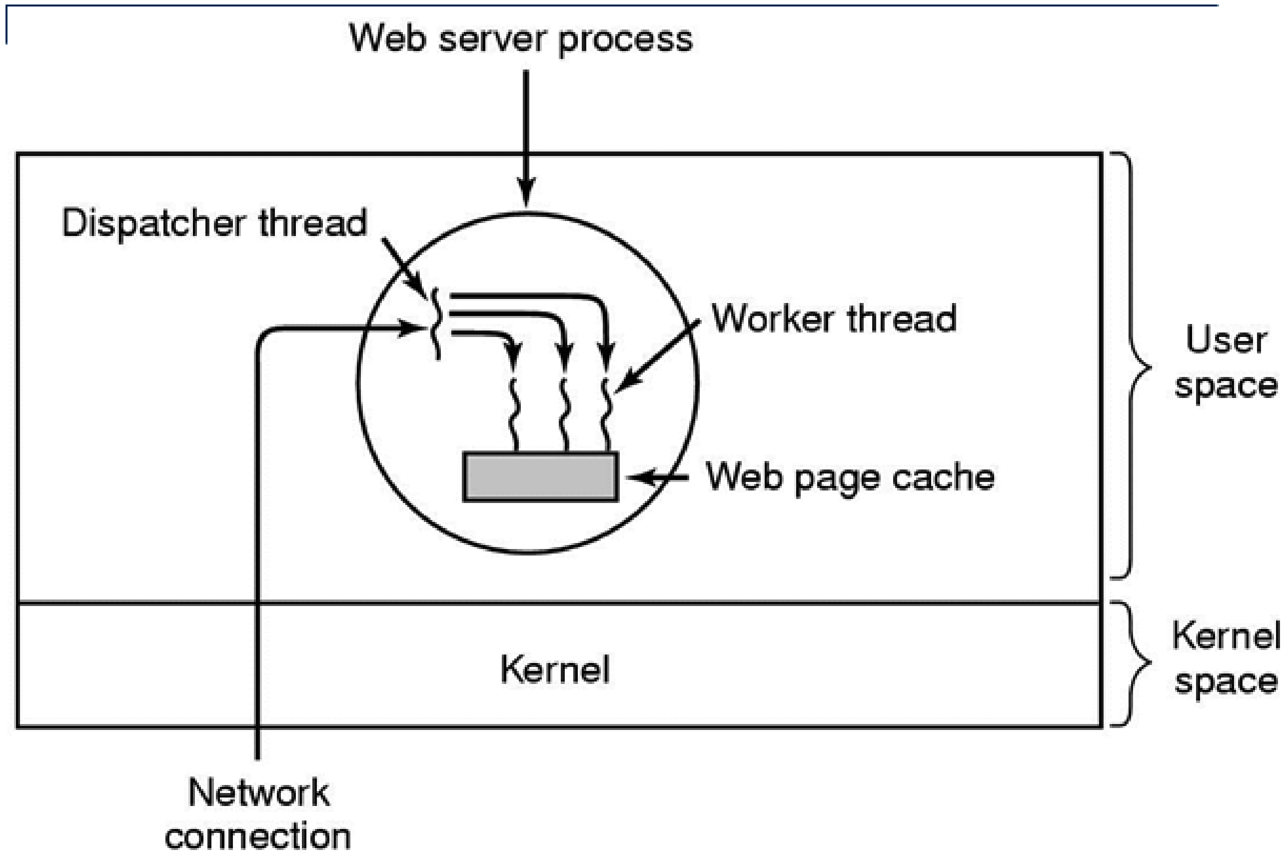
```
while (1) {
  int sock = accept();
  if ((child_pid = fork()) == 0) {
   //Handle client request
   //Close socket and exit
  } else {
   //Close socket
  }
}
```

Threads: Concurrent Servers

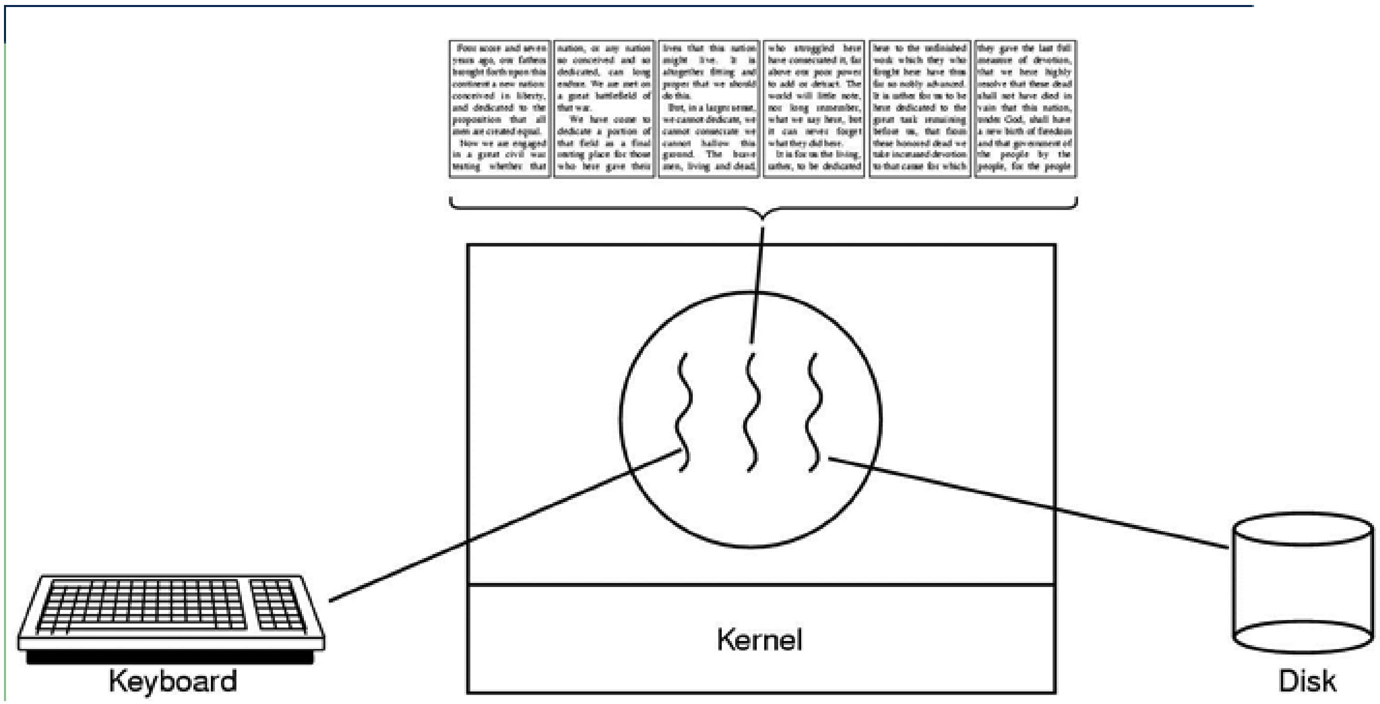
- Instead, we can create a new thread for each request

```
web_server() {
    while (1) {
        int sock = accept();
        thread_create(handle_request, sock);
    }
}
handle_request(int sock) {
       //Process request
       close(sock);
}
```

Thread usage: web server



Thread usage: word processor



- A thread can wait for I/O, while the other threads can still run.
- What if it is single-threaded?

Kernel-Level Threads

- We have taken the execution aspect of a process out and separated it 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 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

Kernel-Level 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
- For such fine-grained concurrency, need even “cheaper” threads

User-Level Threads

- To make threads fast, they need to be implemented at user level
  - Kernel-level threads are managed by the OS
  - User-level threads are managed by the run-time system (user-level library)
- User-level threads are small, cheap, 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
    - No kernel involvement
  - User-level thread operations 100x faster than kernel threads
  - pthreads: PTHREAD_SCOPE_PROCESS

User-Level Thread Limitations

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

Kernel- vs. User-level 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- vs. 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
- Golang today uses user-level threads
  - Multiplex multiple Goroutines (user-level threads) on multiple kernel level threads

Implementing Threads

- Implementing threads has a number of issues
  - Interface
  - Context switch
  - Preemptive vs. Non-preemptive
    - What do they mean?
  - Scheduling
  - Synchronization (next lecture)
- Focus on user-level threads
  - Kernel-level threads are similar to original process management and implementation in the OS

Sample Thread Interface

- thread_create(procedure_t, arg)
  - Create a new thread of control
  - Start executing procedure_t
- thread_yield()
  - Voluntarily give up the processor
- thread_exit()
  - Terminate the calling thread; also thread_destroy
- thread_join(target_thread)
  - Suspend the execution of calling thread until target_thread terminates

Thread Scheduling

- For user-level thread: scheduling occurs entirely in user-space
- 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 (usually one)
- Ready queue: Threads ready to run
- Are there wait queues?

Review of threads

- What are shared among threads of the same process? What are not?
  - Why cannot they share the same stack?
  - How threads of the same process communicate with each other?
- Trade-off between kernel level threads and user level threads
- Blocking system call
  - Blocking system call: an I/O system call that will wait for the I/O to complete before returning
- How do we implement user-level threads

Non-Preemptive Scheduling

- Threads voluntarily give up the CPU with thread_yield

Ping Thread

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

Pong Thread

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

- What is the output of running these two threads?

thread_yield()

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

Implementing thread_yield()

```
thread_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 the magic
  - Saves context of the currently running thread (old_thread)
    - Push all machine state onto its stack (*except stack pointer*)
  - 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

Wait a minute

- Non-preemptive threads have to voluntarily give up CPU
  - Only voluntary calls to thread_yield(), or thread_exit() causes a context switch
- What if a thread never release the CPU (never calls thread_yield())?
- We need preemptive user-level thread scheduling

Preemptive Scheduling

- Preemptive scheduling causes an involuntary context switch
  - Need to regain control of processor asynchronously
- How?
  - *Use timer interrupt*
  - Timer interrupt handler forces current thread to “call” thread_yield
    - How?

Process vs. thread

- Multithreading is only an option for “cooperative tasks”
  - Trust and sharing
- Process
  - Strong isolation but poor performance
- Thread
  - Good performance but share too much
- Example: web browsers
  - Safari: multithreading
    - one webpage can crash entire Safari
  - Google Chrome: each tab has its own process

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

Summary cont.

- But the problem with threads is that, we need to explicitly manage concurrency
  - Why? Access to shared data
  - Synchronization, in the next lecture…
  - It's inherently hard, very hard
- For this and many other reasons, some people argue threads should not be used until absolutely necessary for performance
  - Check out the optional reading by Prof. John Outsterhaut in the 90s…