

Threads Implementation

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
<http://csl.skku.edu>



Today's Topics



- **How to implement threads?**
 - User-level threads
 - Kernel-level threads

- **Threading models**

Kernel/User-level Threads



- **Who is responsible for creating/managing threads?**
 - The OS (kernel threads)
 - Thread creation and management requires system calls
 - The user-level process (user-level threads)
 - A library linked into the program manages the threads

- **Why is user-level thread management possible?**
 - Threads share the same address space
 - The thread manager doesn't need to manipulate address spaces
 - Threads only differ in hardware contexts (roughly)
 - PC, SP, registers
 - These can be manipulated by the user-level process itself.

Kernel-level Threads (1)

■ OS-managed threads

- The OS manages threads and processes.
- All thread operations are implemented in the kernel.
- The OS schedules all of the threads in a system.
 - If one thread in a process blocks (e.g., on I/O), the OS knows about it, and can run other threads from that process.
 - Possible to overlap I/O and computation inside a process.
- Kernel threads are cheaper than processes.
 - Less state to allocate and initialize
- Windows 98/NT/2000/XP/Vista
- Solaris
- Tru64 Unix
- Linux
- Mac OS X

Kernel-level Threads (2)

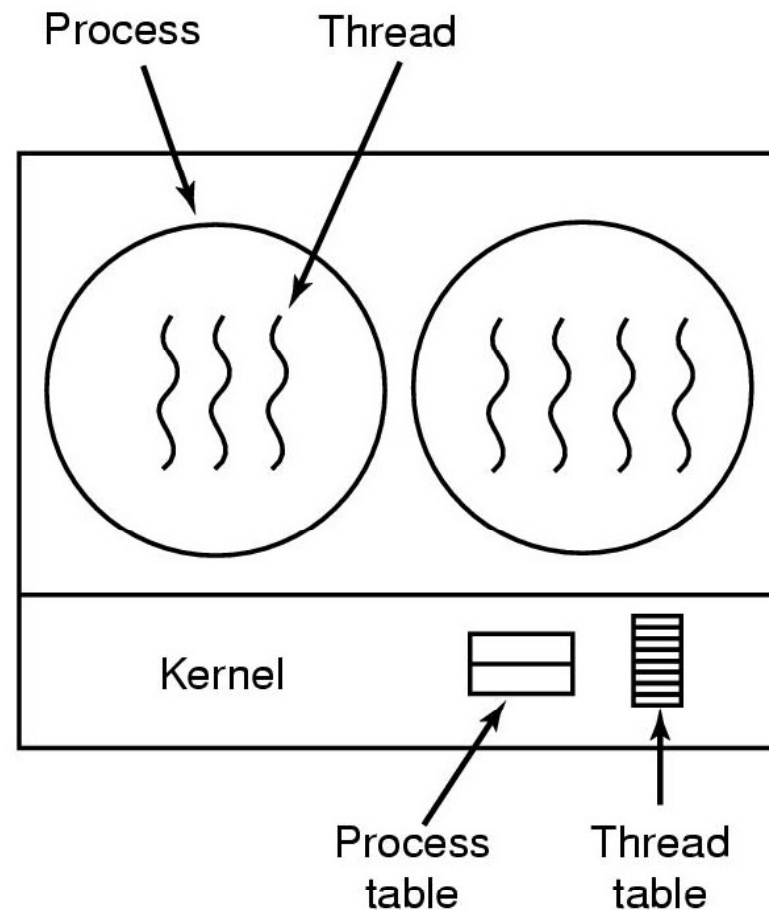
■ Limitations

- They can still be too expensive.
 - For fine-grained concurrency, we need even cheaper threads.
 - Ideally, we want thread operations as fast as a procedure call.
- Thread operations are all system calls.
 - The program must cross an extra protection boundary on every thread operation, even when the processor is being switched between threads in the same address space.
 - The OS must perform all of the usual argument checks.
- Must maintain kernel state for each thread.
 - Can place limit on the number of simultaneous threads. (typically ~1000)
- Kernel-level threads have to be general to support the needs of all programmers, languages, runtime systems, etc.

Implementing Kernel-level Threads

■ Kernel-level threads

- Kernel-level threads are similar to original process management and implementation.



User-level Threads (1)

■ Motivation

- To make threads cheap and fast, they need to be implemented at the user level.
- Portable: User-level threads are managed entirely by the runtime system (user-level library).

■ User-level threads are small and fast

- Each thread is represented simply by a PC, registers, a stack, and a small thread control block (TCB).
- Creating a thread, switching between threads, and synchronizing threads are done via procedure calls (No kernel involvement).
- User-level thread operations can be 10-100x faster than kernel-level threads.

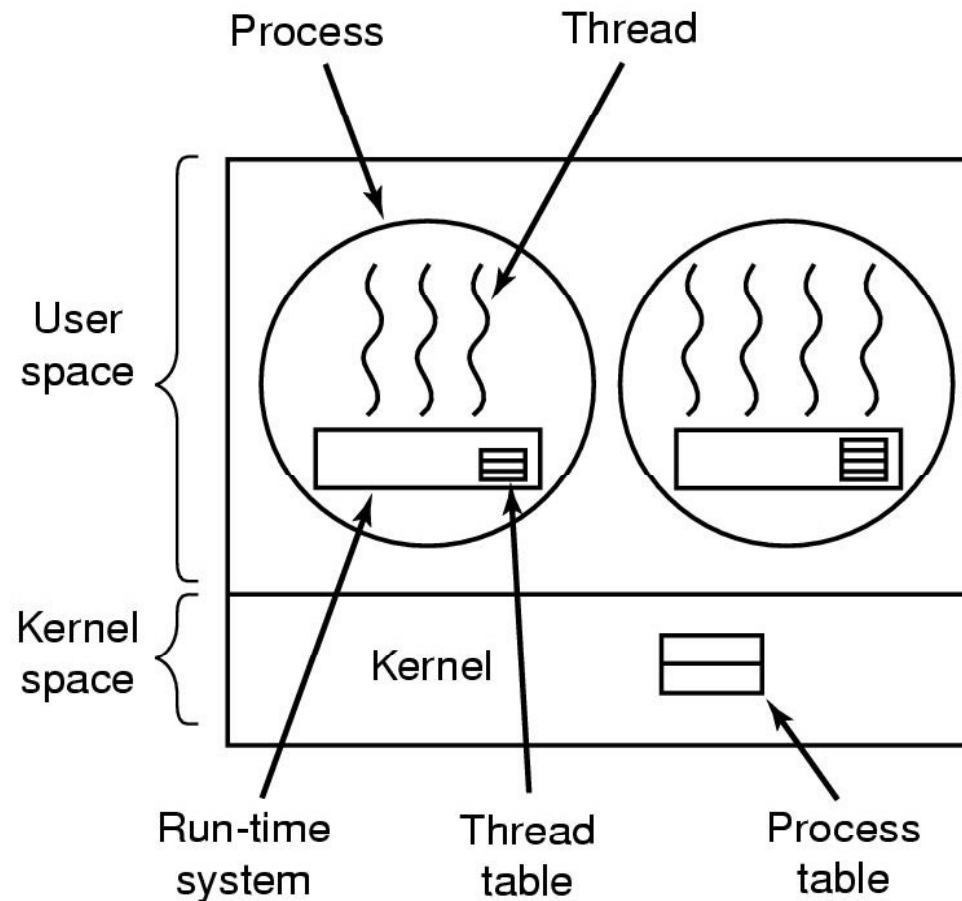
User-level Threads (2)

■ Limitations

- But, user-level threads aren't perfect.
- 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 only idle threads
 - Blocking a process whose thread initiated I/O, even though the process has other threads that are ready to run.
 - Unscheduling a process with a thread holding a lock.
- Solving this requires coordination between the kernel and the user-level thread manager.
 - e.g., all blocking system calls should be emulated in the library via non-blocking calls to the kernel.

Implementing User-level Threads (1)

- User-level threads



Implementing User-level Threads (2)

▪ Thread context switch

- Very simple for user-level threads
 - Save context of currently running thread
 - : push all machine state onto its stack
 - restore context of the next thread
 - : pop machine state from next thread's stack
 - the next thread becomes the current thread
 - return to caller as the new thread
 - : execution resumes at PC of next thread
- All done by assembly languages
 - It works at the level of the procedure calling convention, so it cannot be implemented using procedure calls.

Implementing User-level Threads (3)

▪ Thread scheduling

- A thread scheduler determines when a thread runs.
 - Just like the OS and processes
 - But implemented at user-level as a library
- It uses queues to keep track of what threads are doing.
 - Run queue: threads currently running
 - Ready queue: threads ready to run
 - Wait queue: threads blocked for some reason
(maybe blocked on I/O or a lock)
- How can we prevent a thread from hogging the CPU?

Implementing User-level Threads (4)

■ Non-preemptive scheduling

- Force everybody to cooperate
 - Threads willingly give up the CPU by calling `yield()`.
- `yield()` calls into the scheduler, which context switches to another ready thread.

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

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

- What happens if a thread never calls `yield()`?

Implementing User-level Threads (5)

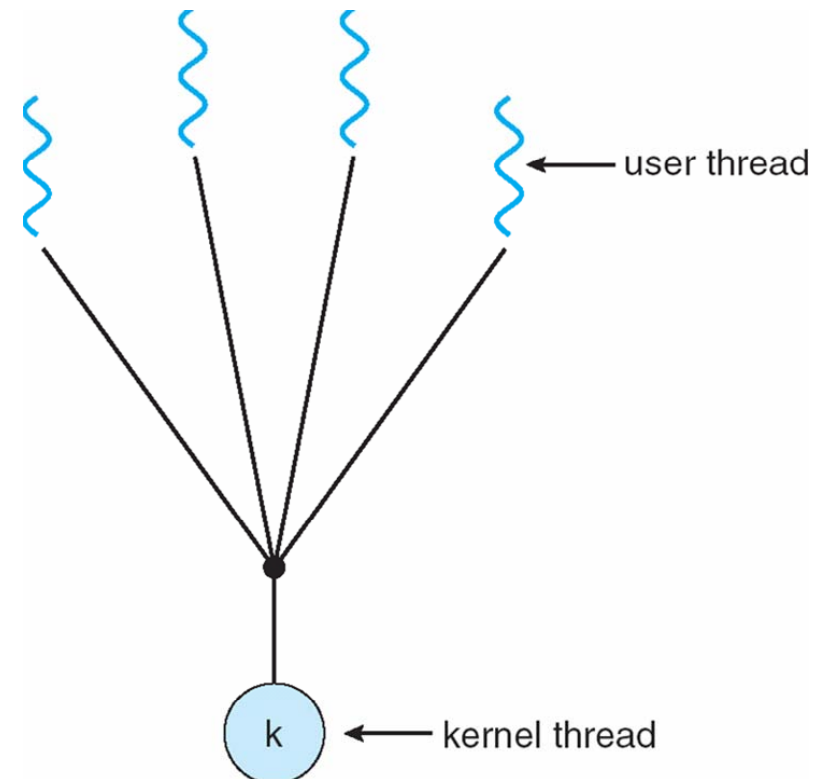
■ Preemptive scheduling

- Need to regain control of processor asynchronously.
- Scheduler requests that a timer interrupt be delivered by the OS periodically.
 - Usually delivered as a UNIX signal
 - Signals are just like software interrupts, but delivered to user-level by the OS instead of delivered to OS by hardware
- At each timer interrupt, scheduler gains control and context switches as appropriate.

Threading Models (1)

■ Many-to-One (N:1)

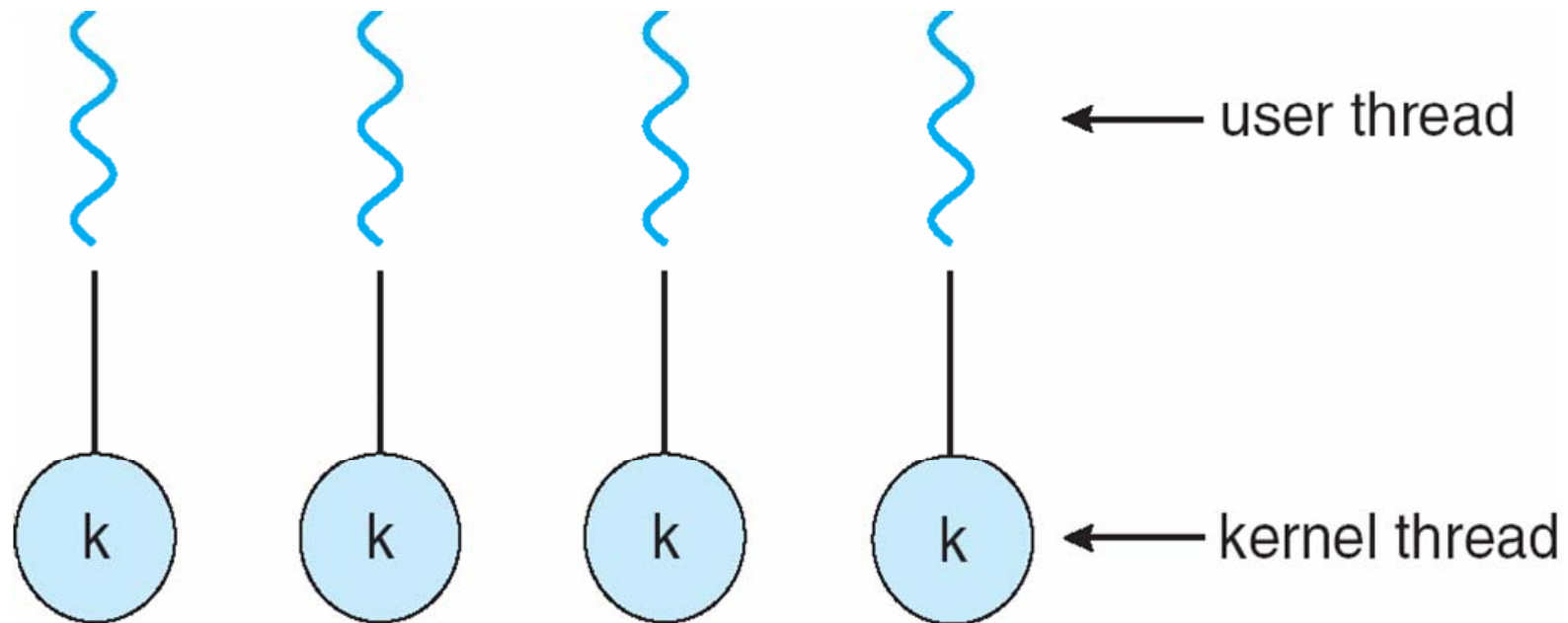
- Many user-level threads mapped to a single kernel thread
- Used on systems that do not support kernel threads.
- Solaris Green Threads
GNU Portable Threads



Threading Models (2)

■ One-to-One (1:1)

- Each user-level thread maps to a kernel thread.
- Windows 98/NT/2000/XP, OS/2, Linux, Solaris 9+



Threading Models (3)

■ Many-to-Many (M:N)

- Allows many user-level threads to be mapped to many kernel threads.
- Allows the OS to create a sufficient number of kernel threads.
- Solaris prior to v9, IRIX, HP-UX, Tru64

