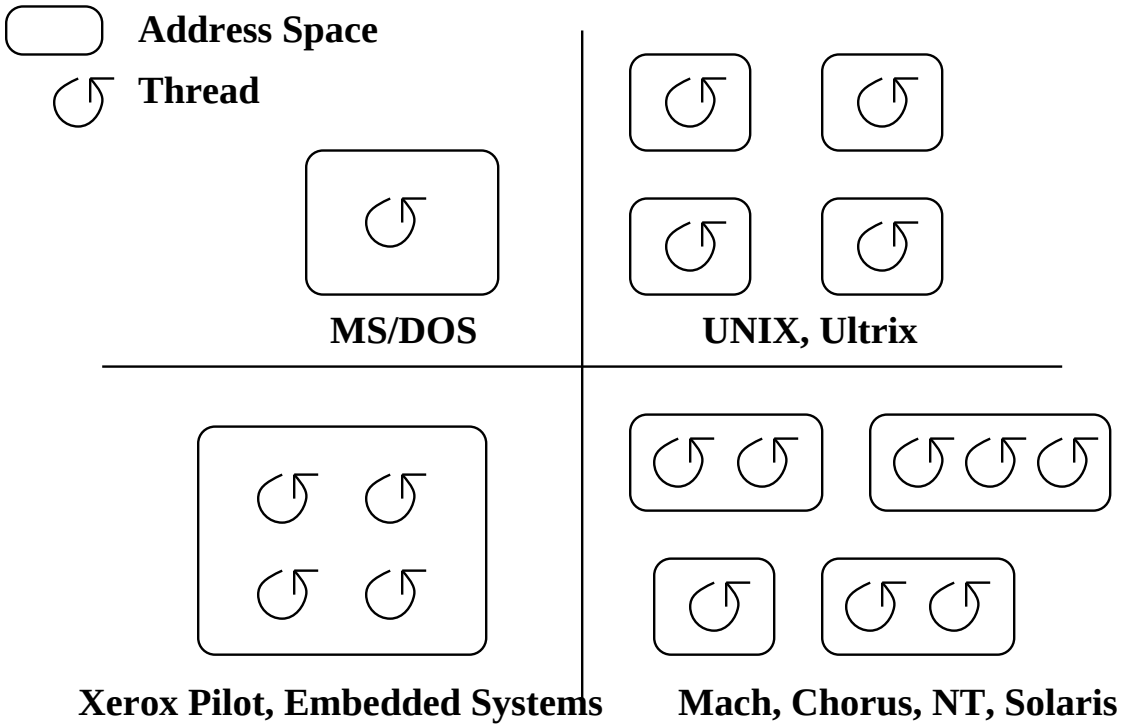
Processes versus Threads *Last Time*

Extract thread of control information from the process, to enable multiple threads to operate within the same process.

A process defines the address space, text, resources, etc.;

A thread defines a single sequential execution stream within a process (PC, SP, registers).

Operating Systems can support one or many address spaces, and one or many threads per address space.



Implementing Threads *Today*

- Where should we implement threads? In the kernel? In a user-level threads package?
- How should we schedule threads (or processes) onto the CPU?

Kernel Threads

A **kernel thread** is a thread that the operating system knows about.

- Switching between kernel threads of the same process requires a mode switch (user-kernel).
  - The values of registers, program counter, and stack pointer must be changed.
  - Memory management information does not need to be changed since the threads share address space.
- The kernel must manage and schedule threads (as well as processes), but it can use the same process scheduling algorithms. (May or may not distinguish processes and threads.)

$\Longrightarrow$ Switching between kernel threads is faster than switching between processes.

User-Level Threads

A **user-level thread** is a thread that the OS does *not* know about.

- The OS only knows about the existence of the process containing the threads.
- The OS only schedules the process, not the threads within the process.
- The programmer uses a *thread library* to manage threads (create and delete them, synchronize them, and schedule them).
- There is no mode switch involved when switching threads.
- User-level thread scheduling is more flexible, since the user-level code can define a problem dependent thread scheduling policy.
  - Each process might use a different scheduling algorithm for its own threads.
  - A thread can voluntarily give up the processor by telling the scheduler it will *yield* to other threads. (But!)

$\Longrightarrow$ User-level threads are typically much faster than kernel threads since they do not require system calls to create them or mode switches to move between them.

Kernel Threads vs. User-Level Threads: performance

Ultrix: 1 thread per address space

Topaz: multiple kernel threads per address space

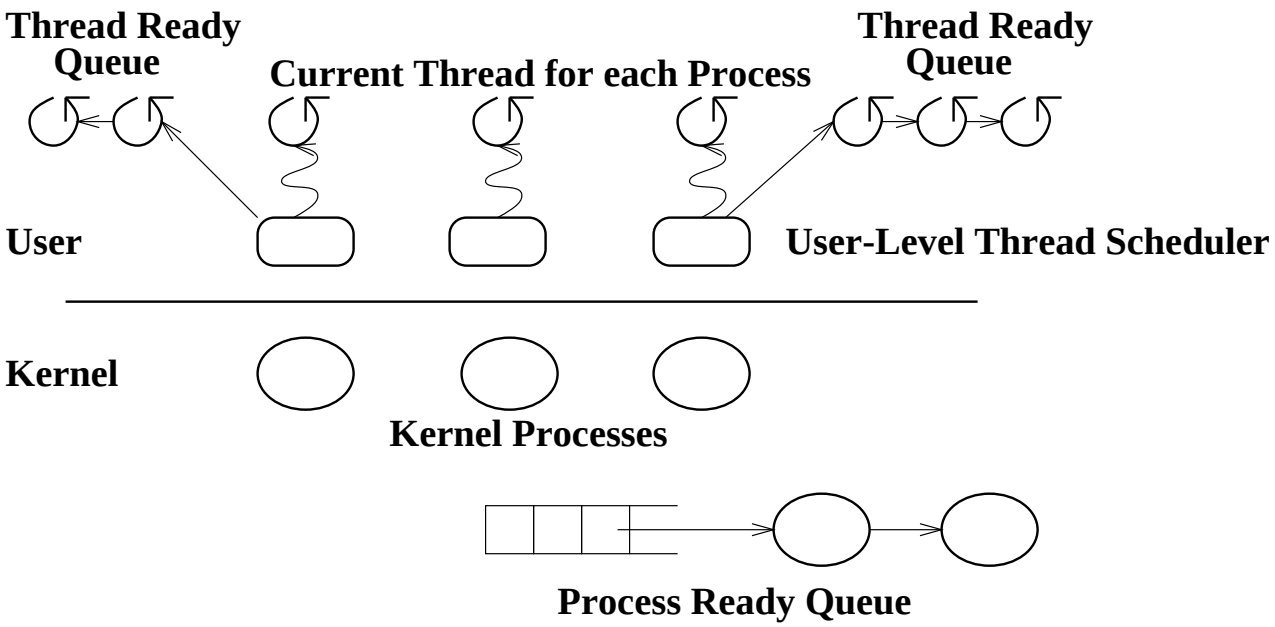
FastThreads: multiple user threads per address space

	Ultrix	Topaz	FastThreads
Fork	11,320	1208	39
Signal/Wait	1846	229	52

Operation times in Microseconds on a MIPS 3000

$\Longrightarrow$ User-level thread operations are orders of magnitude faster than similar kernel thread operations,

User-Level Threads



User-Level Threads

The major advantage of user-level threads is improved performance compared to kernel threads, but there are several disadvantages:

- Since the OS does not know about the existence of the threads, it may make poor scheduling decisions:
  - It might run a process that only has idle threads.
  - If a user-level thread is waiting for I/O, the entire process (that is, all the threads) will wait.

$\Longrightarrow$ Solving this problem requires communication between the kernel and the user-level thread manager.

Related Issues:

2. For kernel threads, the more threads a process creates, the more time slices the OS will dedicate to it.
3. For user-level threads, the OS just knows about the process; regardless of the number of user threads, the OS time slices the process the same way with all the other processes.

Kernel Threads and User-Level Threads in Solaris

Summary

- Switching between user-level threads is faster than between kernel threads since a mode switch is not required.
- User-level threads may result in the kernel's making poor scheduling decisions, resulting in slower process execution than if kernel threads were used.