Last Class: CPU Scheduling

- Pre-emptive versus non-preemptive schedulers
- Goals for Scheduling:
  - Minimize average response time
  - Maximize throughput
  - Share CPU equally
  - Other goals?
- **Scheduling Algorithms:**
  - Selecting a scheduling algorithm is a policy decision - consider tradeoffs
  - FSCS
  - Round-robin
  - SJF/SRTF
  - MLFQ
  - Lottery scheduler

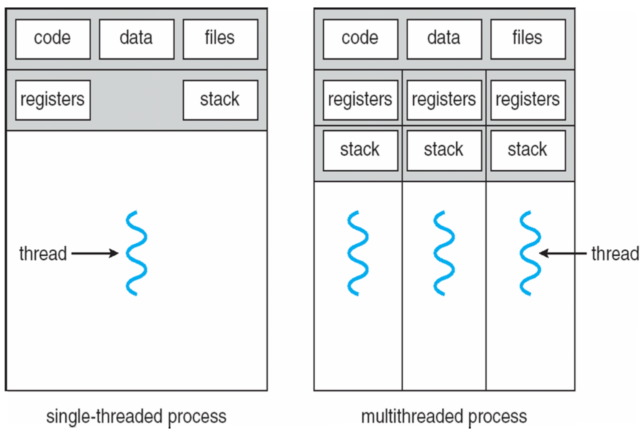
Today: Threads

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

Processes versus Threads

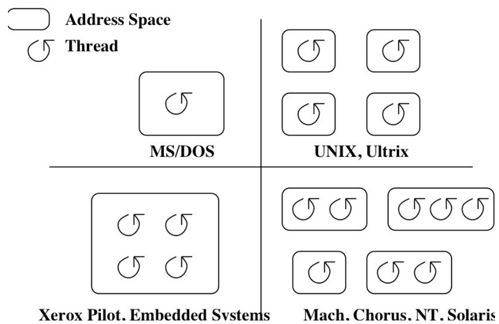
- **A process** defines the address space, text, resources, etc.,
- **A thread** defines a single sequential execution stream within a process (PC, stack, registers).
- Threads extract the *thread of control* information from the process
- Threads are bound to a single process.
- Each process may have multiple threads of control within it.
  - The address space of a process is shared among all its threads
  - No system calls are required to cooperate among threads
  - Simpler than message passing and shared-memory

Single and Multithreaded Processes



Classifying Threaded Systems

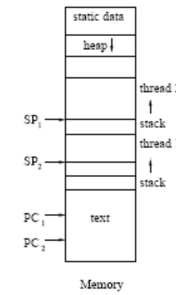
Operating Systems can support one or many address spaces, and one or many threads per address space.



Example Threaded Program

```
main()
    global in, out, n, buffer[n];
    in = 0; out = 0;
    fork_thread (producer());
    fork_thread (consumer());
end
producer
    repeat
        nextp = produced item
        while in+1 mod n = out do no-op
        buffer[in] = nextp; in = (in+1) mod n
consumer
    repeat
        while in = out do no-op
        nextc = buffer[out]; out = (out+1) mod n
        consume item nextc
```

One possible memory layout:



- Forking a thread can be a system call to the kernel, or a procedure call to a thread library (user code).

Kernel Threads

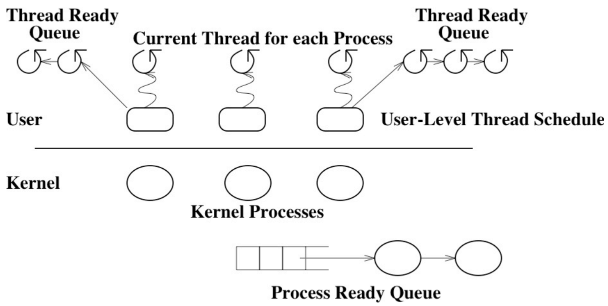
- **A kernel thread**, also known as a **lightweight process**, is a thread that the operating system knows about.
- Switching between kernel threads of the same process requires a small context switch.
  - The values of registers, program counter, and stack pointer must be changed.
  - Memory management information does not need to be changed since the threads share an address space.
- The kernel must manage and schedule threads (as well as processes), but it can use the same process scheduling algorithms.

➔Switching between kernel threads is slightly 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 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).

User-Level Threads



User-Level Threads: Advantages

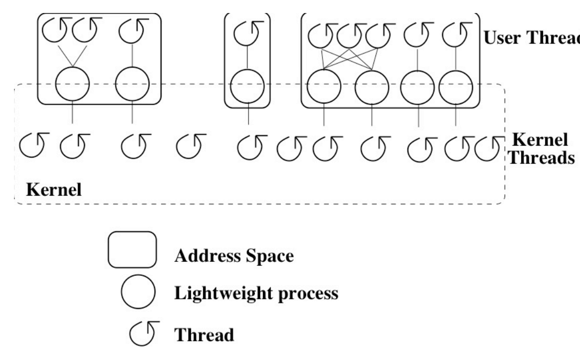
- There is no context switch involved when switching threads.
- User-level thread scheduling is more flexible
  - A user-level code can define a problem dependent thread scheduling policy.
  - Each process might use a different scheduling algorithm for its own threads.
  - A thread can voluntarily give up the processor by telling the scheduler it will *yield* to other threads.
- User-level threads do not require system calls to create them or context switches to move between them

➔ User-level threads are typically much faster than kernel threads

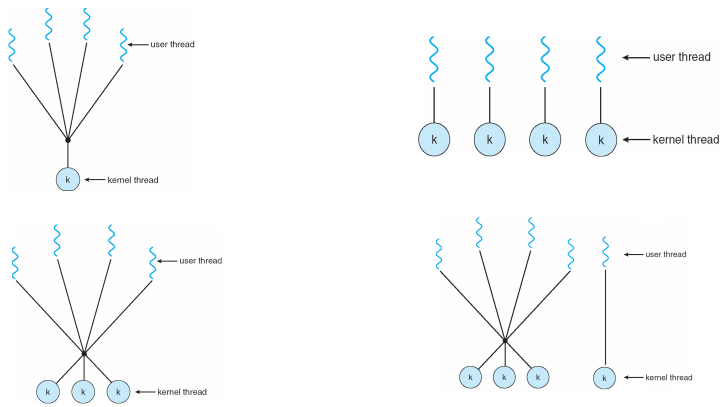
User-Level Threads: Disadvantages

- Since the OS does not know about the existence of the user-level 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 will wait.
  - Solving this problem requires communication between the kernel and the user-level thread manager.
- Since the OS just knows about the process, it schedules the process the same way as other processes, regardless of the number of user threads.
- For kernel threads, the more threads a process creates, the more time slices the OS will dedicate to it.

Example: Kernel and User-Level Threads in Solaris

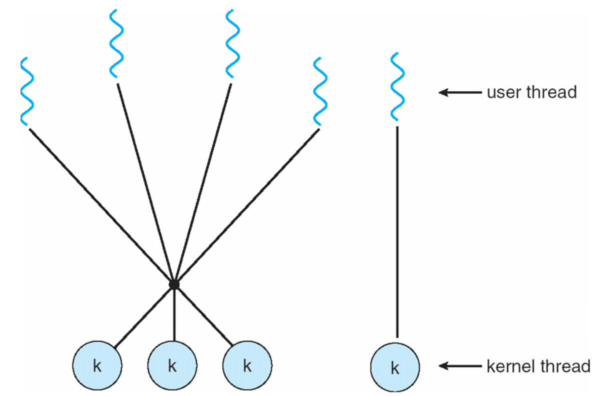


Threading Models



- Many-to-one, one-to-one, many-to-many and two-level

Two-level Model



Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)
- WIN32 Threads: Similar to Posix, but for Windows

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

Examples

```
Pthreads:
    pthread_attr_init(&attr);  /* set default attrributes */
    pthread_create(&tid, &attr, sum, &param);

Win32 threads
ThreadHandle  = CreateThread(NULL, 0, Sum, &Param, 0, &ThreadID);


Java Threads:

Sum sumObject = new Sum();
Thread t = new Thread(new Summation(param, SumObject));
t.start();  // start the thread
```


Summary

- Thread: a single execution stream within a process
- Switching between user-level threads is faster than between kernel threads since a context switch is not required.
- User-level threads may result in the kernel making poor scheduling decisions, resulting in slower process execution than if kernel threads were used.
- Many scheduling algorithms exist. Selecting an algorithm is a policy decision and should be based on characteristics of processes being run and goals of operating system (minimize response time, maximize throughput, ...).