CIS 452 - Operating Systems Concepts

Nathan Bowman

Images taken from Silberschatz book

Scheduling Algorithms -- Threads and Multiprocessor Systems

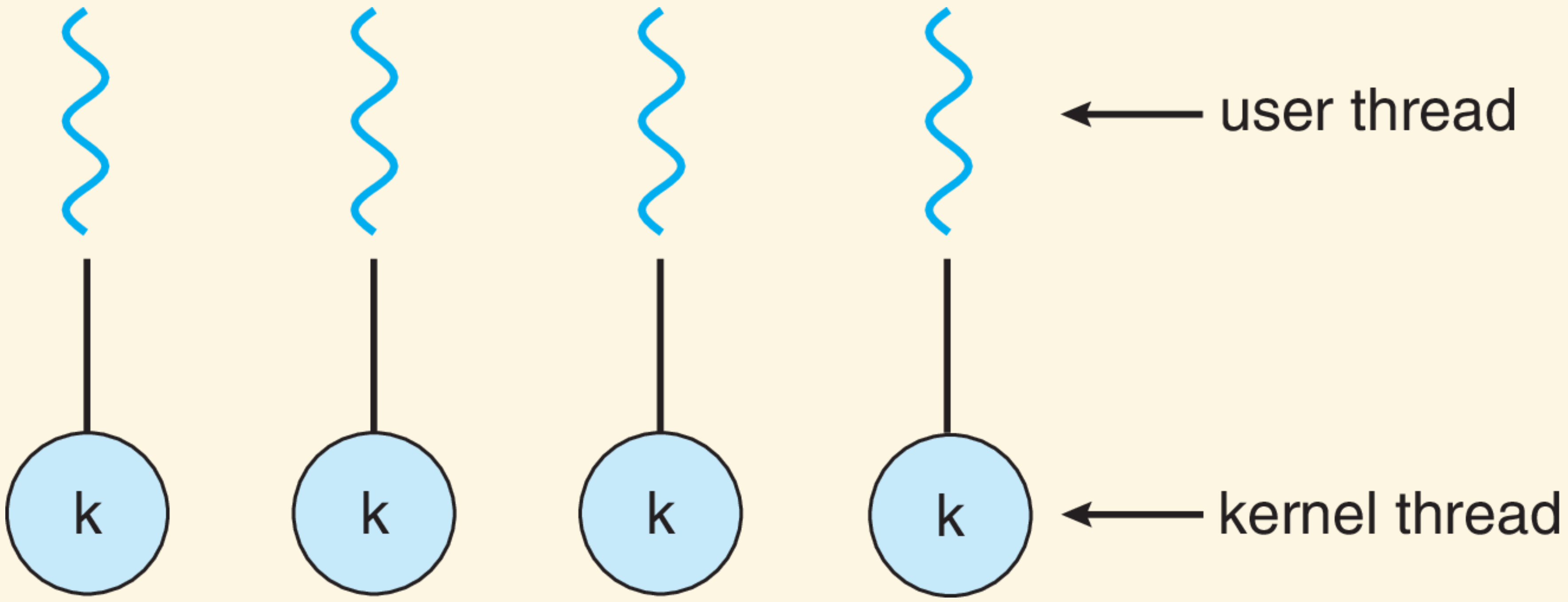
Up until now, most of what we have disussed could apply equally well to threads or processes

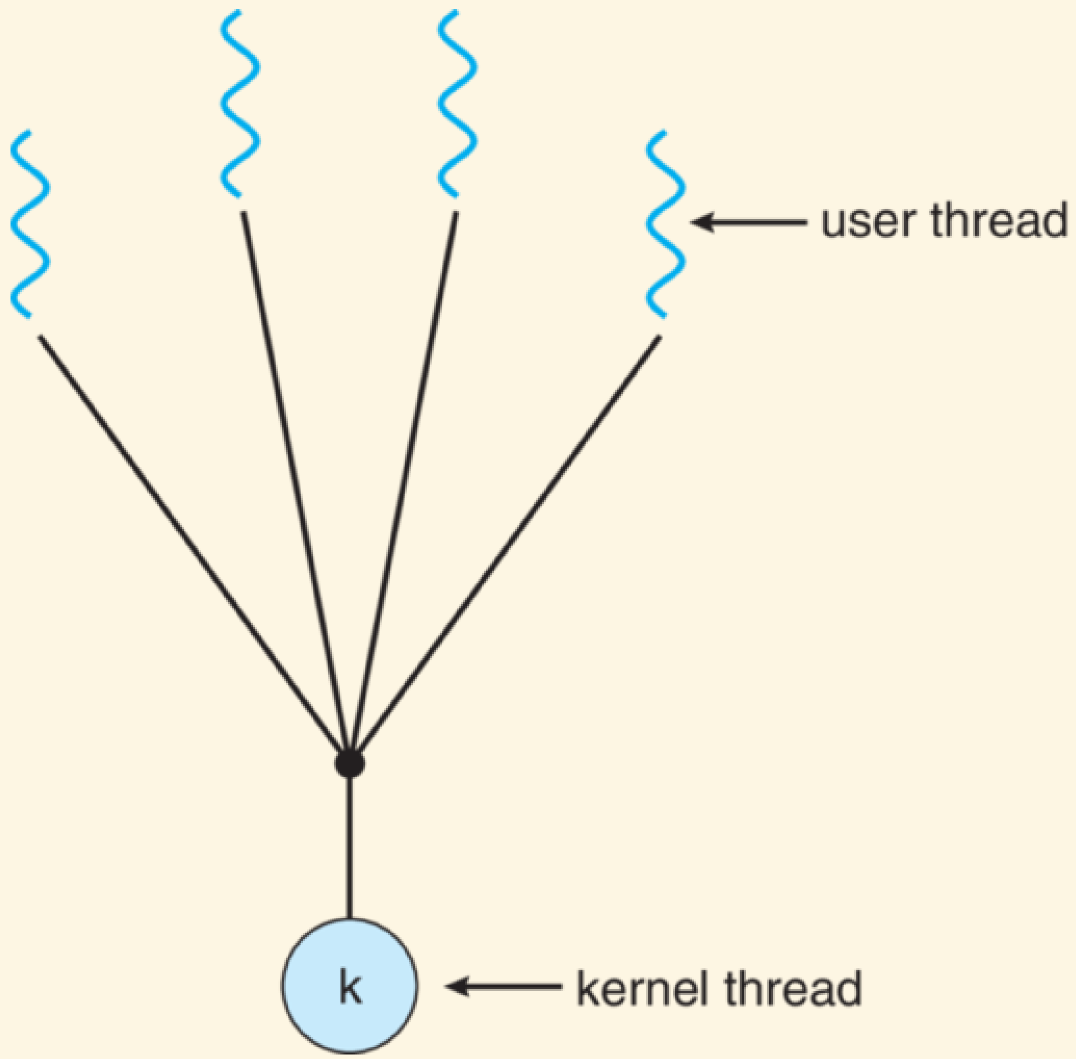
There are additional considerations that can arise when scheduling threads

First, recall the distinction user threads (seen only by the threading library) and kernel threads (managed by the OS)

Sometimes, there is just one user thread per kernel thread (a 1:1 model), in which case there is no difference

Other times, there may be more than one user thread per kernel thread (a many-to-one model)


user thread
k
k
k
k
kernel thread


user thread
k
kernel thread

Contention scope is the level at which threads are competing for the CPU

In a many-to-one model, user-level threads have **process-contention scope** (PCS)

Only one thread in the process can be scheduled at once, and the threading library must make a choice

Typically done by a fixed priority that can be controlled by the user

All kernel-level threads have **system-contention scope** (SCS)

Each thread competes with all other threads on the system for CPU time

This may use any of the scheduling algorithms mentioned previously

Because threading is 1:1 in Windows and Linux, all scheduling is done via SCS

Multiprocessor Scheduling

There are also new issues that arise when scheduling for more than one processor

We will make the simplifying assumption that all processors in the system have the same characteristics/capabilities

First question that must be answered: who will control the scheduling for the different processors?

Asymmetric multiprocessing -- a single main processor handles all scheduling decisions (along with several other OS functions)

This has the benefit of simplicity: because only one processor is accessing system data structures, we are not as concerned with race conditions in the kernel

Symmetric multiprocessing (SMP) is more efficient and harder to do

Each processor is responsible for its own scheduling

Could be the case that

- all processes are in a shared queue, or
- each processor has a private queue of processes to run

Various race conditions apply

Windows, Linux, Mac OS X, and most other modern OSes employ SMP

Another issue that can impact the efficiency of multiprocessor systems is **processor affinity**

When a process is running on a processor, its most important values from memory are placed in the cache for that processor

Each processor has its own cache

If a process moves to another processor, it essentially has its cache flushed

SMP systems should thus try to avoid moving processes between processors

We say that a processor has an "affinity" for the processor it is currently running on

A system supports **soft affinity** if it attempts to keep a process on the same CPU but does not guarantee it will do so

Hard affinity is a guarantee that a process will remain on the same CPU

Many systems support both types, sometimes allowing the user to make a system call specifying a particular type

The final issue we discuss related to multiprocessor scheduling is **load balancing**

It is inefficient for a CPU to be idle while another CPU has more work than it can handle

Load balancing is attempting to keep the work split as evenly as possible among CPUs in a system

This is only an issue if each processor has a private queue of processes to run

Otherwise, an idle processor will simply grab a process from the shared queue

Most SMP systems *do* have private queues

Push migration is when a specific task runs periodically and moves processes from overloaded CPUs to CPUs that are less busy

Pull migration is when an idle CPU "pulls" a task from a busy one

Combination of both is often used

Note that whether it is pushed or pulled, a process is being moved between processors, which will result in its cache being invalidated

When to migrate a process (i.e., how imbalanced the load must be) is a design decision that will vary between systems