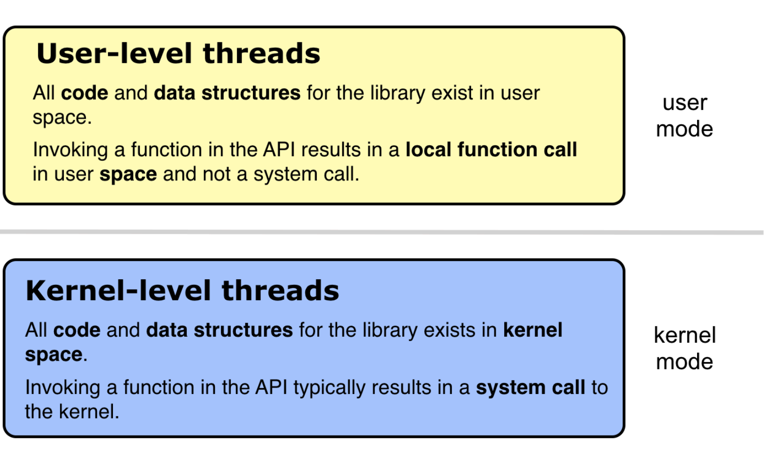
Operating systems
Spring 2020
Search...
Home
Site navigation
Modules
Learning and examination
Course material
Grading
Pairs and groups
Timeline and schedule
0 - Preparations
1 - Fundamental concepts
2 - The process

Implementing threads

A thread library provides programmers with an API for creating and managing threads. Support for threads must be provided either at the user level or by the kernel.

- Kernel level threads are supported and managed directly by the operating system.
- User level threads are supported above the kernel in user space and are managed without kernel support.



Kernel level threads

Kernel level threads are supported and managed directly by the operating system.

- The kernel knows about and manages all threads.
- One process control block (PCP) per process.
- One thread control block (TCB) per thread in the system.
- Provide system calls to create and manage threads from user space.

Advantages

- The kernel has full knowledge of all threads.
- Scheduler may decide to give more CPU time to a process having a large numer of threads.
- Good for applications that frequently block.

Disadvantages

- Kernel manage and schedule all threads.
- Significant overhead and increase in kernel complexity.
- Kernel level threads are slow and inefficient compared to user level threads.
- Thread operations are hundreds of times slower compared to user-level threads.

User level threads

User level threads are supported above the kernel in user space and are managed without kernel support.

- Threads managed entirely by the run-time system (user-level library).
- Ideally, thread operations should be as fast as a function call.
- The kernel knows nothing about user-level threads and manage them as if they where single-threaded processes.

Advantages

- Can be implemented on an OS that does not suport kernel-level threads.
- Does not require modifications of the OS.
- Simple representation: PC, registers, stack and small thread control block all stored in the user-level process address space.
- Simple management: Creating, switching and synchronizing threads done in user-space without kernel intervention.
- Fast and efficient: switching threads not much more expensive than a function call.

Disadvantages

- Not a perfect solution (a trade off).
- Lack of coordination between the user-level thread manager and the kernel.
- OS may make poor decisions like:
  - scheduling a process with idle threads
  - blocking a process due to a blocking thread even though the process has other threads that can run
  - giving a process as a whole one time slice irrespective of whether the process has 1 or 1000 threads
  - unschedule a process with a thread holding a lock.
- May require communication between the kernel and the user-level thread manager (scheduler activations) to overcome the above problems.

User-level thread models

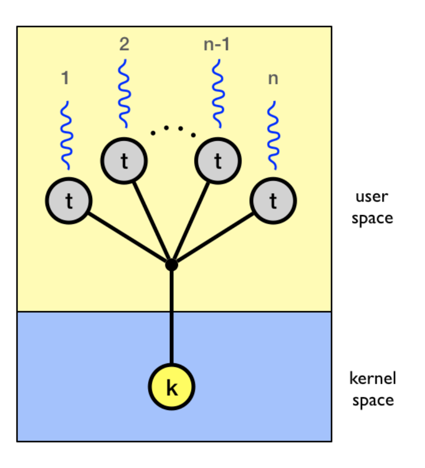
In general, user-level threads can be implemented using one of four models.

- Many-to-one
- One-to-one
- Many-to-many
- Two-level

All models maps user-level threads to kernel-level threads. A **kernel thread** is similar to a process in a non-threaded (single-threaded) system. The kernel thread is the unit of execution that is scheduled by the kernel to execute on the CPU. The term **virtual processor** is often used instead of kernel thread.

Many-to-one

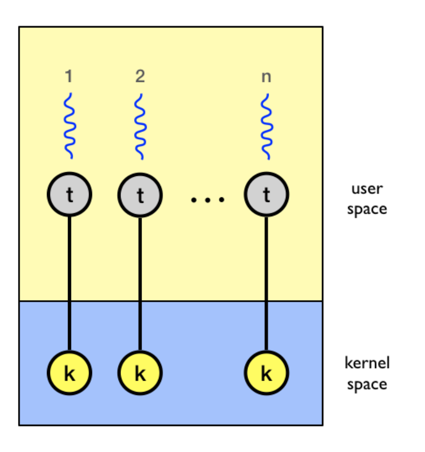
In the many-to-one model all user level threads execute on the same kernel thread. The process can only run one user-level thread at a time because there is only one kernel-level thread associated with the process.



The kernel has no knowledge of user-level threads. From its perspective, a process is an opaque black box that occasionally makes system calls.

One-to-one

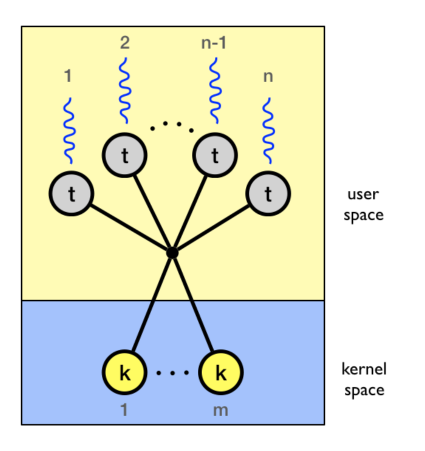
In the one-to-one model every user-level thread execute on a separate kernel-level thread.



In this model the kernel must provide a system call for creating a new kernel thread.

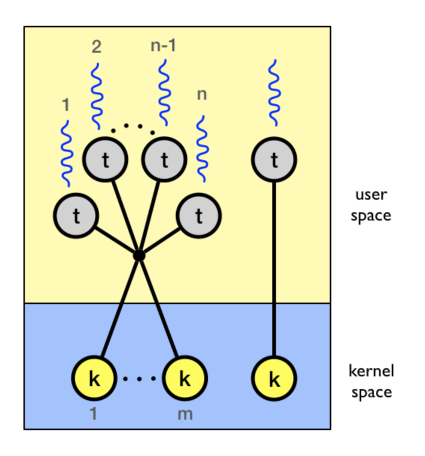
Many-to-many

In the many-to-many model the process is allocated m number of kernel-level threads to execute n number of user-level thread.



Two-level

The two-level model is similar to the many-to-many model but also allows for certain user-level threads to be bound to a single kernel-level thread.



Scheduler activations

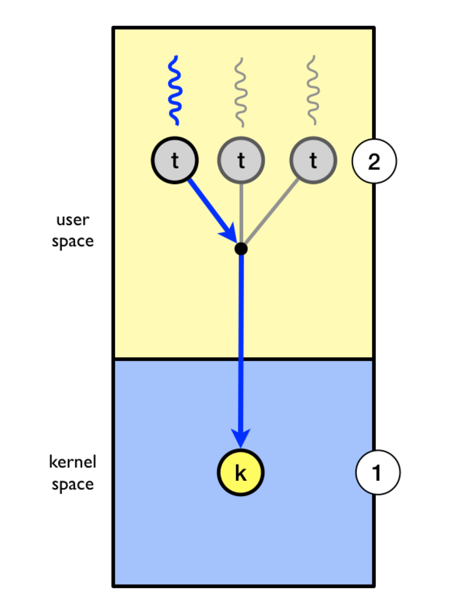
In both the many-to-many model and the two-level model there must be some way for the kernel to communicate with the user level thread manager to maintain an appropriate number of kernel threads allocated to the process. This mechanism is called **scheduler activations**.

The kernel provides the application with a set of kernel threads (virtual processors), and then the application has complete control over what threads to run on each of the kernel threads (virtual processors). The number of kernel threads (virtual processors) in the set is controlled by the kernel, in response to the competing demands of different processes in the system.[1]

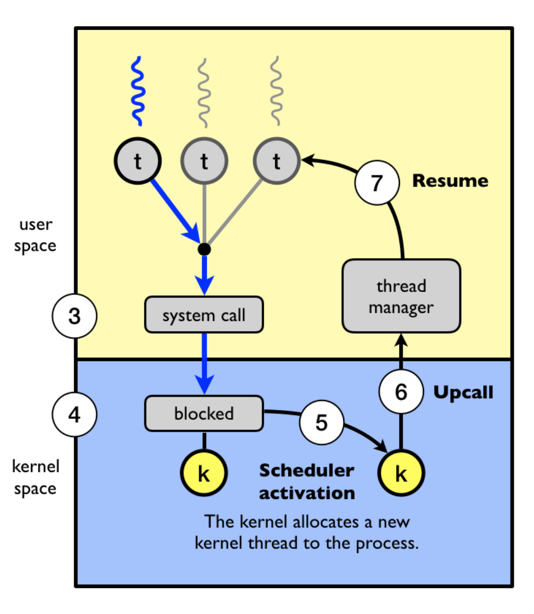
The kernel notify the user-level thread manager of important kernel events using **upcalls** from the kernel to the user-level thread manager. Examples of such events includes a thread making a blocking system call and the kernel allocating a new kernel thread to the process.[1]

Example

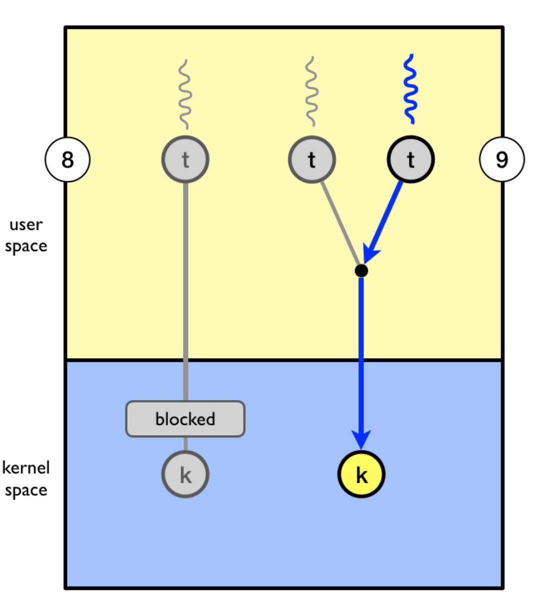
Let's study an example of how scheduler activations can be used. The kernel has allocated **one kernel thread** (1) to a process with **three user-level threads** (2). The three user level threads take turn executing on the single kernel-level thread.



The executing thread makes a **blocking system call** (3) and the the kernel blocks the calling user-level thread and the kernel-level thread used to execute the user-level thread (4). **Scheduler activation:** the kernel decides to allocate a **new kernel-level thread** to the process (5). **Upcall:** the kernel **notifies** the user-level **thread manager** which user-level thread that is now blocked and that a new kernel-level thread is available (6). The user-level thread manager move the other threads to the new kernel thread and resumes one of the ready threads (7).



While one user-level thread is blocked (8) the other threads can take turn executing on the new kernel thread (9).



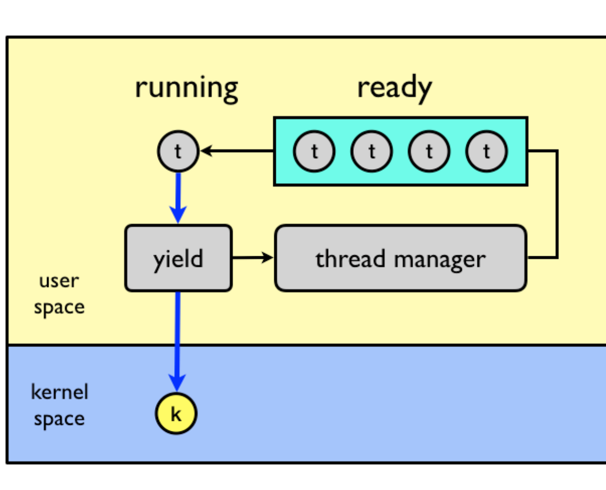
User-level thread scheduling

Scheduling of the usea-level threads among the available kernel-level threads is done by a thread scheduler implemented in user space. There are two main methods: cooperative and preemptive thread scheduling.

Cooperative scheduling of user-level threads

The cooperative model is similar to multiprogramming where a process executes on the CPU until making a I/O request. Cooperative user-level threads execute on the assigned kernel-level thread until they voluntarily give back the kernel thread to the thread manager.

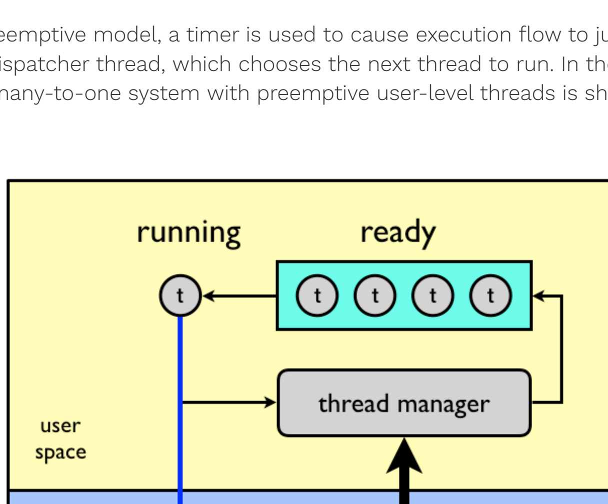
In the cooperative model, threads **yield** to each other, either explicitly (e.g., by calling a `yield()` provided by the user-level thread library) or implicitly (e.g., requesting a lock held by another thread). In the below figure a many-to-one system with cooperative user-level threads is shown.



Preemptive scheduling of user-level threads

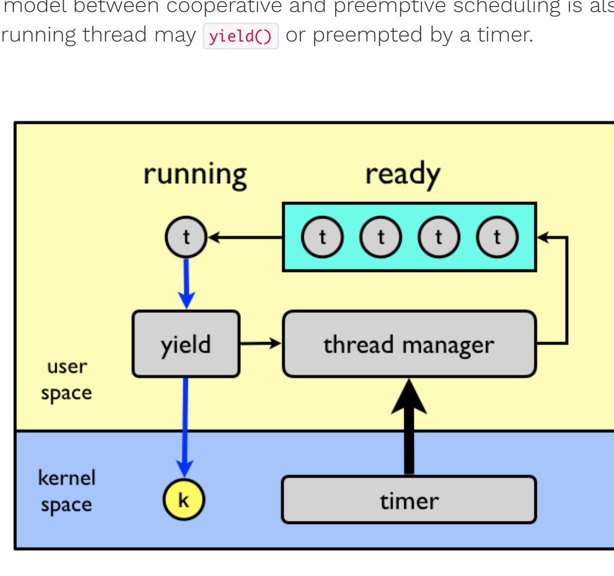
The preemptive model is similar to multitasking (aka time sharing). Multitasking is a logical extension of multiprogramming where a timer is set to cause an interrupt at a regular time interval and the running process is replaced if the job requests I/O or if the job is interrupted by the timer. This way, the running job is given a time slice of execution than cannot be exceeded.

In the preemptive model, a timer is used to cause execution flow to jump to a central dispatcher thread, which chooses the next thread to run. In the below figure a many-to-one system with preemptive user-level threads is shown.



Cooperative and preemptive (hybrid) scheduling of user-level threads

A hybrid model between cooperative and preemptive scheduling is also possible where a running thread may `yield()` or preempted by a timer.



1. An Implementation of Scheduler Activations on the NetBSD Operating System ↩