ITP30002 Operating System

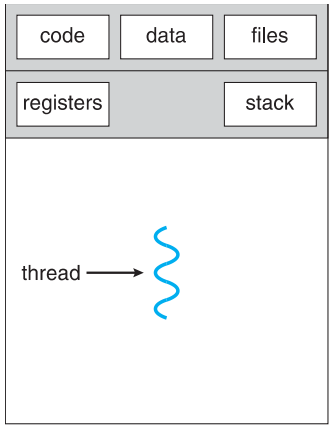
Multithreading

This lecture note is taken from the instructor's resource of Operating System Concept, 10/e and then partly edited/revised by Shin Hong.

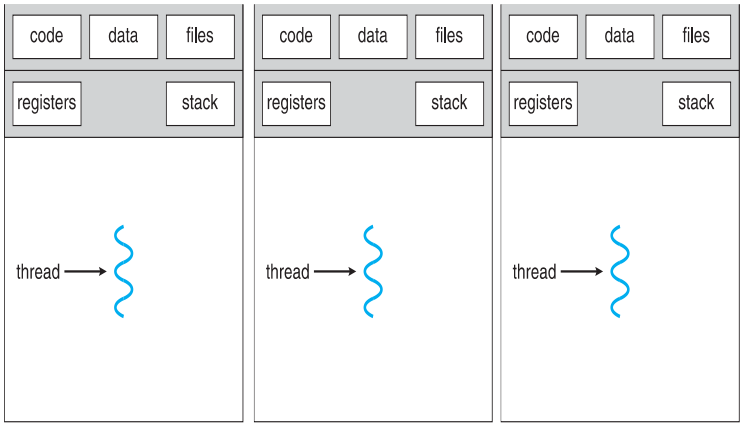
What is Multithreading?

- A thread is a single flow of control
- Threads run within an application
- Most modern applications are multithreaded to run multiple tasks within an application
  - E.g., a web brower does update display, fetch data, spell checking, and answer a network request at the same time within an application

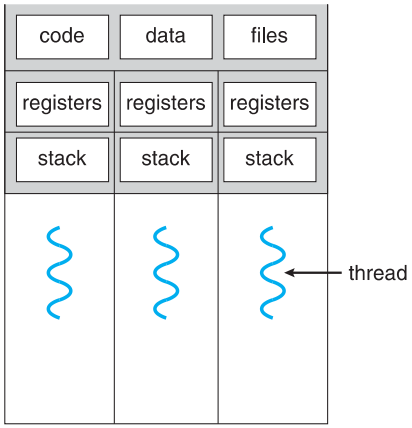
Single and Multithreaded Processes



<Single threaded process>



<Multi-processing>

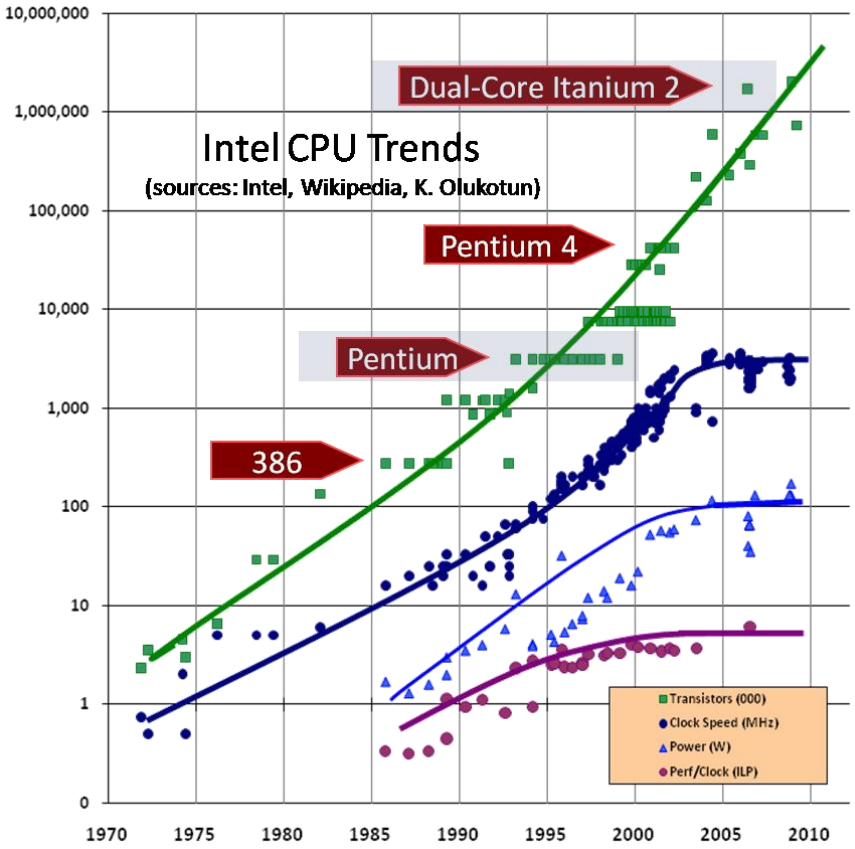


<Multi-threaded process>

Benefits

- Scalability: process can take advantage of multiprocessor architectures
- Responsiveness: may allow continued execution if part of process is blocked, especially important for user interfaces
- Resource sharing: threads share resources of process, easier than shared memory or message passings
- Efficiency: cheaper than process creation, thread switching lower overhead than context switching

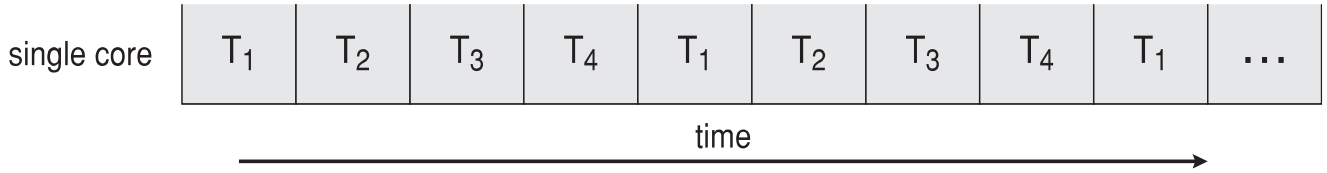
Why Concurrent Programming



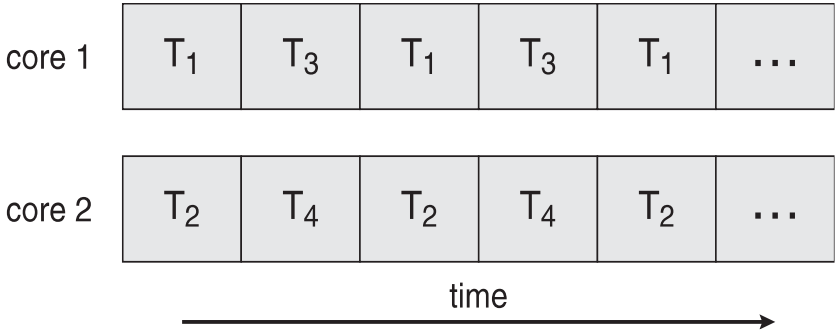
- *Parallelism* implies a system can perform more than one task simultaneously
- *Concurrency* supports more than one task making progress simultaneously
  - Single processor/core, scheduler providing concurrency
- Multicore or multiprocessor systems putting pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging

Concurrency vs. Parallelism

Concurrent execution on single-core system:



Parallelism on a multi-core system:



Multicore Programming (Cont.)

- Types of parallelism
  - Data parallelism – distributes subsets of the same data across multiple cores, same operation on each
  - Task parallelism – distributing threads across cores, each thread performing unique operation

- As the number of threads grows in programs, so does architectural support for threading
  - CPUs have cores as well as *hardware threads*

Amdahl's Law

- Upper bound of performance gains by adding additional cores to an application that has both serial and parallel components
  - *S* is serial portion
  - *N* processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

  - That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
  - As *N* approaches infinity, speedup approaches 1 / *S*

- Serial portion of an application has disproportionate effect on performance gained by adding additional cores

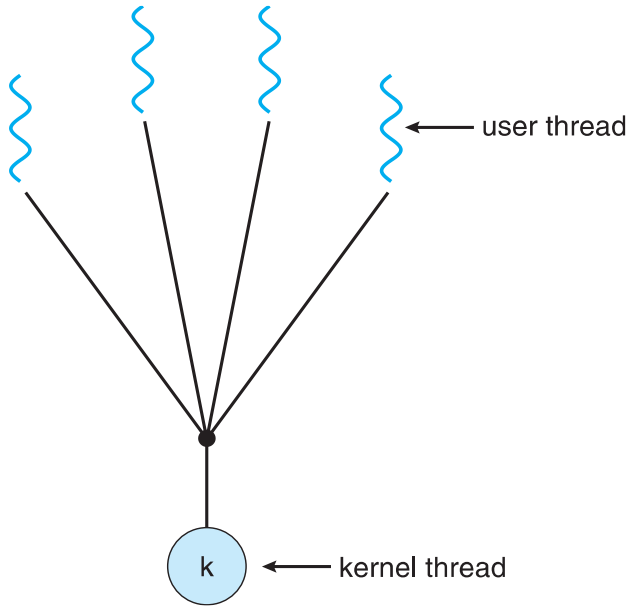
User Threads and Kernel Threads

- User threads: management done by user-level threads library
  - POSIX Pthreads
  - Windows threads
  - Java threads

- Kernel threads: supported by the Kernel
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

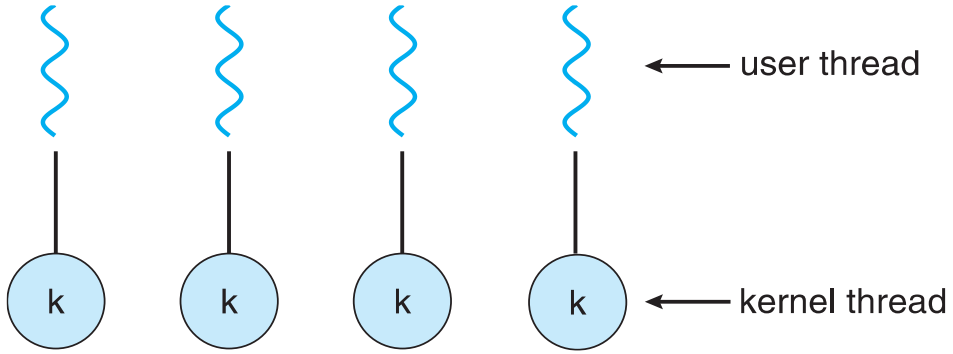
Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
  - Solaris Green Threads
  - GNU Portable Threads



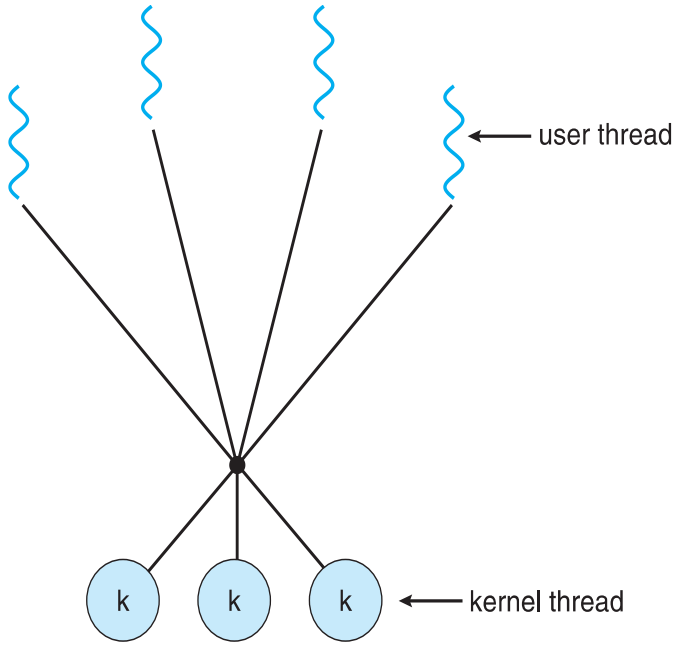
One-to-One

- Each user-level thread maps to a kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
  - Windows
  - Linux
  - Solaris 9 and later



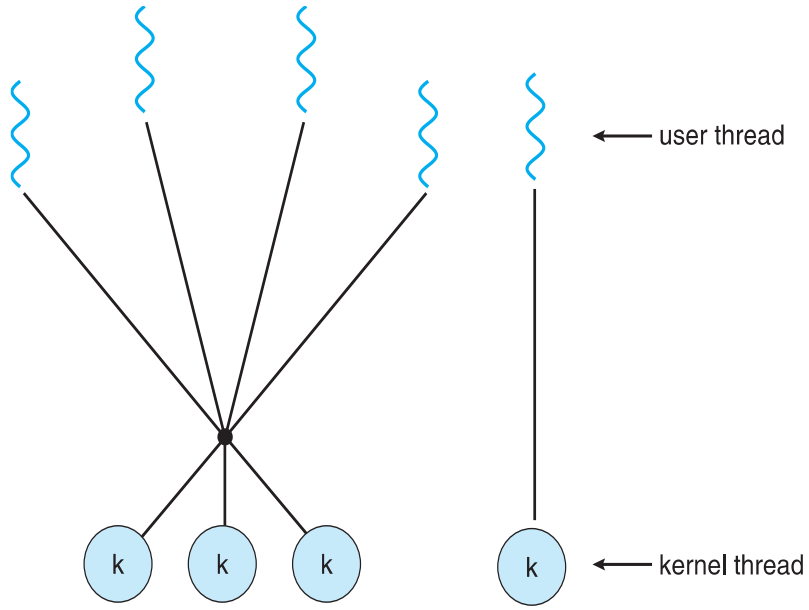
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
  - Solaris prior to version 9
  - Windows with the *ThreadFiber* package



Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



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

Pthread (POSIX Thread)

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
  - Specification, not implementation;
  - API specifies behavior of the thread library, implementation is up to development of the library
  - May be provided either as user-level or kernel-level

- Common in UNIX operating systems

Implicit Threading

- Growing in popularity as the number of threads increases, ensuring program correctness is more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
  - OpenMP
  - Thread Pools
  - Grand Central Dispatch
  - Microsoft Threading Building Blocks (TBB)
  - `java.util.concurrent` package

OpenMP

- A set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel
  - run for loop in parallel
  - create as many threads as there are cores
  - ex.

```
#pragma omp parallel for for(i=0;i<N;i++) {
    c[i] = a[i] + b[i];
}
```

Thread Pools

- Create a number of threads in a pool where they await work
- Advantages
  - Slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
  - Separating parallel tasks from threading mechanisms

Threading Design Issue: Fork & Exec

- Does `fork()` duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork
- `exec()` usually works as to replace the running process including all threads

Threading Design Issue: Signal

- Where should a signal be delivered for multi-threaded?
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process

Threading Design Issue: Thread Cancellation

- Terminating a thread before it has finished
- Two general approaches
  - Asynchronous cancellation terminates the target thread immediately
  - Deferred cancellation allows the target thread to periodically check if it should be cancelled
    - a thread terminates next time when it reaches at a predefined cancellation point

Threading Design Issue: Thread-Local Storage

- Thread-local storage (TLS) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to `static` data
  - TLS is unique to each thread

Threading Design Issue: Scheduler Activations

- Problem: Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Solution: use Light Weight Process (LWP), an intermediate data structure between user and kernel threads
  - Appears to be a virtual processor on which process can schedule user thread to run
  - Each LWP attached to kernel thread
  - Kernel raises upcalls to user-level threading library to give the information on kernel threads for the threading library to maintain the correct number kernel threads