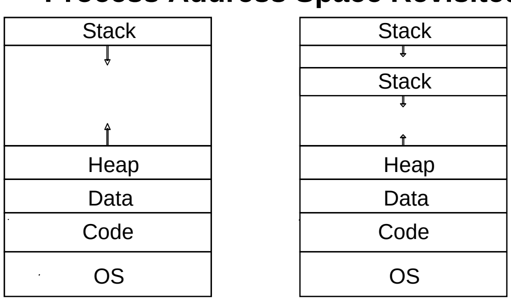
Multithreaded Programming

- Overview
- Multithreading Models
- Thread Libraries
- Threading Issues

What are Threads?

- Thread
  - Independent stream of instructions
  - Basic unit of CPU utilization
- A thread contains
  - A thread ID
  - A register set (including the Program Counter PC)
  - An execution stack
- A thread shares with its sibling threads
  - The code, data and heap section
  - Other OS resources, such as open files and signals

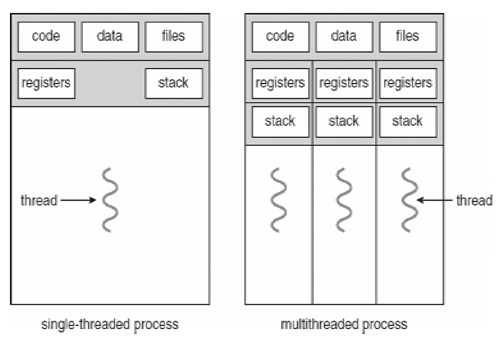
Process Address Space Revisited



(a) Process with Single Thread

(b) Process with Two Threads

Single and Multithreaded Processes



Multi-Threaded Processes

- Each thread has a private stack
- But threads share the process address space!
- There’s no memory protection!
- Threads could potentially write into each other’s stack

Why use Threads?

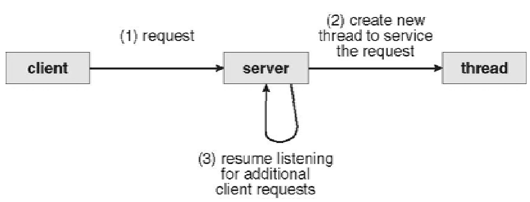
- A specific example, a Web server:

```
do
{       get web page request from client
        check if page exists and client has permissions
        transmit web page back to client
} while(1);
```

- If transmission takes very long time, server is unable to answer other client’s requests. Solution:

```
do
{       get web page request from client
        check if page exists and client has permissions
        create a thread to transmit web page back to client
} while(1);
```


Multithreaded Server Architecture



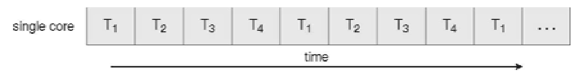
Benefits of Multithreaded Programming

- Responsiveness
- Resource Sharing
- Economy
- Scalability

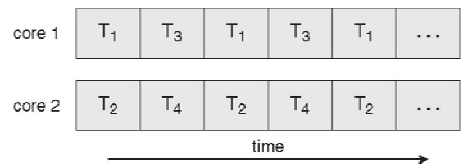
Multicore Programming

- Multicore systems putting pressure on programmers, challenges include
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**

Multicore Programming



Concurrent Execution on a Single-core System



Parallel Execution on a multicore System

User Threads

- Thread management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

Kernel Threads

- Supported by the Kernel
- Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

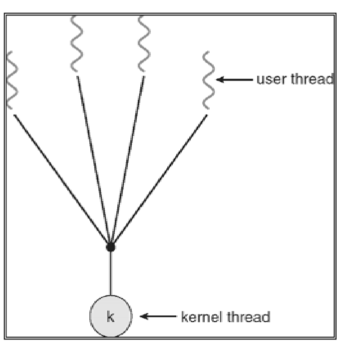
Multithreading Models

- Relationships between user threads and kernel threads
  - Many-to-One
  - One-to-One
  - Many-to-Many

Many-to-One

- Many user-level threads mapped to single kernel thread
  - Thread management by thread library in user space → efficient
  - No concurrency
  - Unable to run in parallel on multiprocessors (MP)
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

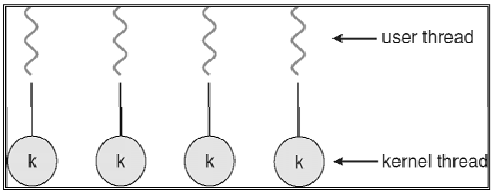
Many-to-One Model



One-to-One

- Each user-level thread maps to a kernel thread
  - More concurrency
  - Multiple threads to run in parallel on multiprocessors
  - Overhead of creating kernel threads
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

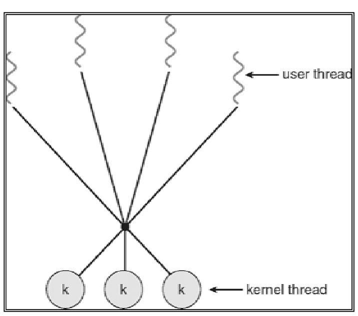
One-to-one Model



Many-to-Many Model

- Allows many user level threads to be mapped to a smaller or equal number of kernel threads
  - Concurrency
  - Can run in parallel on a multiprocessor
  - Developers can create as many user threads as necessary
- Examples
  - Solaris prior to version 9
  - Windows NT/2000 with the *ThreadFiber* package

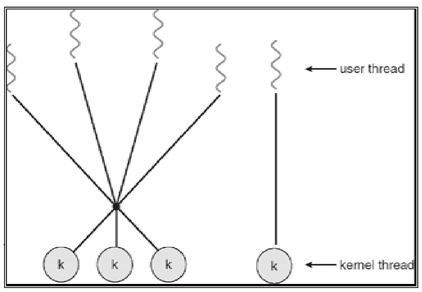
Many-to-Many Model



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

Two-level Model



Thread Libraries

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

Pthreads

- Refers to the POSIX standard (IEEE 1003.1c)
- API for thread creation and synchronization
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)
- May be provided either as user-level or kernel-level

Java Threads

- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface
- JVM manages Java threads
  - Creation
  - Execution
  - Etc.

Practice Exercise

- Consider two threads sharing a global variable count, initially 10:

Thread A	Thread B
`count++;`	`count--;`

- What are the possible values for count after both threads finish executing:

Practice Exercise

- Consider the following three concurrent threads that share a global variable g, initially 10:

Thread A	Thread B	Thread C
`g = g * 2;`	`g = g + 1;`	`g = g - 2;`

- What are the possible values for g, after all three threads finish executing?

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Thread cancellation
- Signal handling
- Thread pools
- Thread specific data
- Scheduler activations

Semantics of fork() and exec()

- Does **fork()** duplicate only the calling thread or all threads?
  - Which of the two versions of fork() to use depends on the application
    - exec() immediately after fork()
    - No exec() after fork()

Thread Cancellation

- Terminating a thread before it has completed
- A thread that is cancelled is called target thread
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Problems: when resources have been allocated to a canceled thread or while in the midst of updating data sharing with other threads

Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
  - A **signal handler** is used to process signals. All signals follow this pattern
    1. Signal is generated by particular event
    2. Signal is delivered to a process
    3. Once delivered, the Signal must be handled
- Synchronous vs. asynchronous signals
- A signal may be handled by one of following handlers:
  - Default signal handlers
  - user-defined handlers

Signal Handling

- Options to deliver signals in multithreaded programs:
  - 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
- ex:
  - UNIX function for delivering a signal is kill(pid_t pid, int signal)
  - POSIX thread provides the pthread_kill(pthread_t tid, int signal)

Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually 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
- Ex:
  - QueueUserWorkItem(LPTHREAD_START_ROUTINE Function, PVOID Param, ULONG Flags) in Win32 API
  - java.util.concurrent package in Java 1.5

Thread-Specific Data

- 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)

Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the thread library
  - An intermediate data structure called *LWP* (Lightweight Process) between user thread and kernel thread
- This communication allows an application to maintain the correct number of kernel threads

Operating System Examples

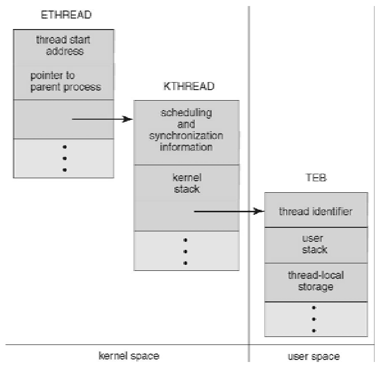
- Windows XP Threads
- Linux Threads

Windows XP Threads

- Implements the one-to-one mapping
- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area

  Context of the threads
- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

Windows XP Threads



Linux Threads

- Linux refers to them as *tasks* rather than process or *thread*
- Thread creation is done through **clone()** system call
- **clone()** allows a child task to share the address space of the parent task (process)
  - CLONE_FS, CLONE_VM, CLONE_SIGHAND, CLONE_FILES

Linux Threads

- Linux provides clone() system call to create threads
  - When clone() is invoked, it is passed set of flags, which determine how much sharing is to take place between parent and child tasks

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Java Thread States