Chapter 4: Threads

Chapter 4: Threads

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples

Objectives

- To introduce the notion of a thread—a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To explore several strategies that provide implicit threading
- To examine issues related to multithreaded programming
- To cover operating system support for threads in Windows and Linux

Motivation

- **A thread** is a basic unit of CPU utilization; it comprises a *thread ID*, a *program counter*, a *register set*, and a *stack*.
  - Most modern applications are multithreaded
  - Thread that run within an application shares with other threads belonging to the same process
- **Process creation** is *heavy-weight* while **thread creation** is *light-weight*
  - A traditional ***process*** has a single thread of control.
    - If a process has multiple threads of control, it can perform more than one task at a time.
  - **Figure 4.1** illustrates the difference between a traditional **single-threaded process** and a **multithreaded process.**

Motivation

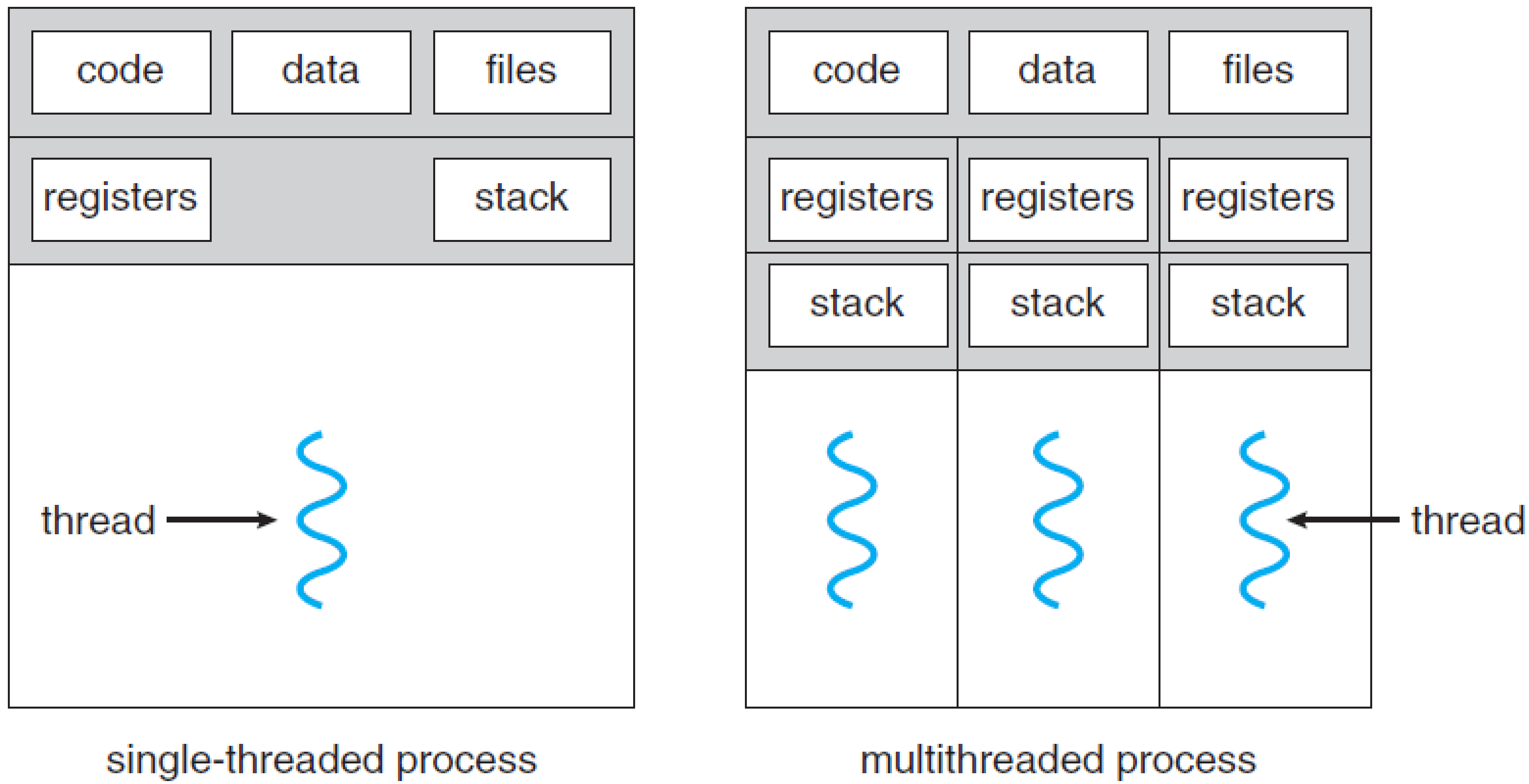


Figure 4.1 Single-threaded and multithreaded processes.

Motivation

- Most software applications that run on modern computers are **multithreaded**;
  - An application typically is implemented as a **separate process** with **several threads** of control.
    - For example, a web browser might have one thread display images or text while another thread retrieves data from the network.
- Applications designed to leverage processing capabilities on **multicore** systems.
  - Such applications can perform **several CPU-intensive tasks in parallel** across the **multiple computing cores**.

Motivation

- It is generally more efficient to use one process that contains **multiple threads**;
  - If the web-server process is **multithreaded**, the server will create a separate thread that listens for client requests.
  - When a request is made, rather than creating another process, the server creates a new thread to service the request and resume listening for additional requests.
    - This is illustrated in Figure 4.2.
- Threads also play a vital role in **remote procedure call** (RPC) systems.

Motivation

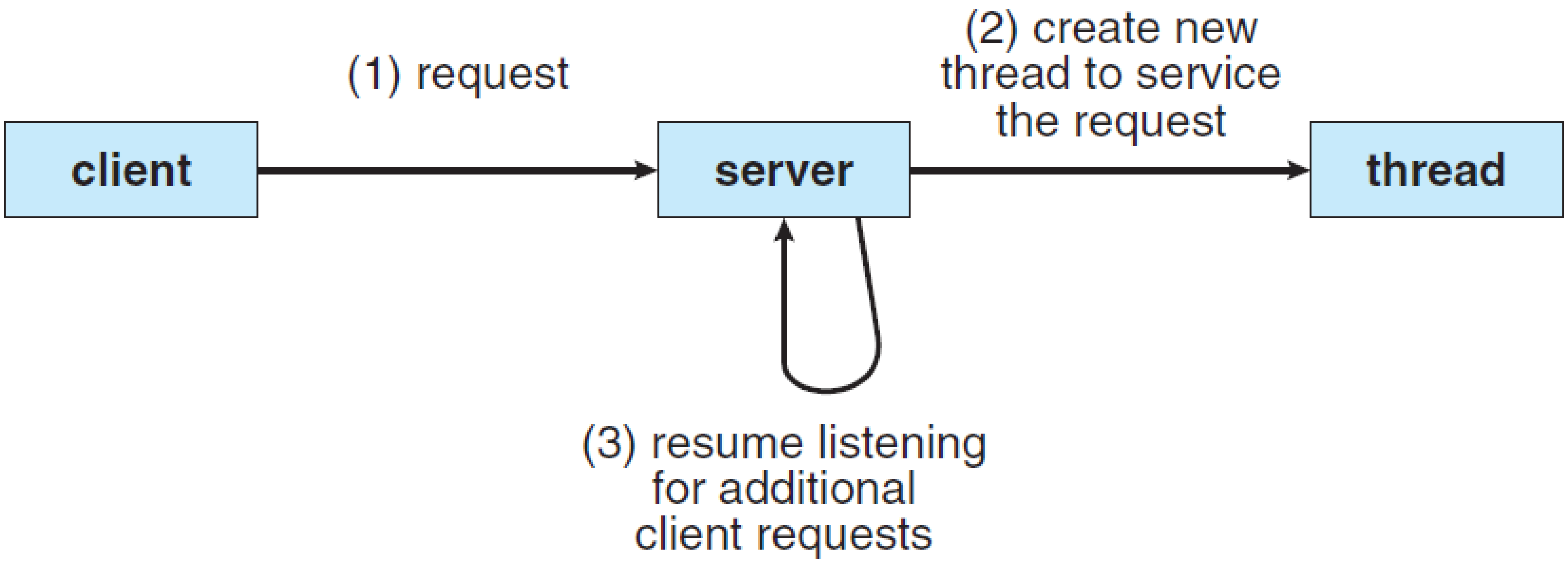


Figure 4.2 Multithreaded server architecture.

Motivation

- Most OS kernels are now multithreaded.
- Several threads operate in the **kernel**, and each thread performs a specific task, such as *managing devices*, *managing memory*, *interrupt handling*, etc.
  - For example,
    - **Solaris** has a set of threads in the kernel specifically for interrupt handling;
    - **Linux** uses a kernel thread for managing the amount of free memory in the system.
    - **Windows** has many threads for supporting its various functions

Benefits of Multithreaded Programming

- The benefits of **multithreaded programming** can be broken down into **four** major categories:
  - **Responsiveness:** Multithreading an interactive application allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing *responsiveness* to the user.
    - This quality is especially useful in designing user interfaces.
  - **Resource sharing:** Processes can only share resources through techniques such as ***shared memory*** and ***message passing***.
    - The benefit of <u>sharing code and data</u> is that it allows an application to have several different threads of activity *within the same address space* (memory location).

Benefits of Multithreaded Programming

- **Economy:** Allocating memory and resources for process creation is costly.
  - it is more economical to create and *context-switch* threads.
- **Scalability:** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores.
  - A single-threaded process can run on only one processor, regardless how many are available.

Multicore Programming

- A more recent trend in system design is to place multiple computing cores on a single chip.
- Each core appears as a **separate processor** to the OS and call these **multi-core** or **multiprocessor** systems.
  - **Multithreaded programming** provides a mechanism for more efficient use of these **multiple cores**.
  - Consider an application with **four threads** on a system with a **Single Processor**, concurrency (*happening at the same time*) merely means that the execution of the threads will be **interleaved over time** (see **Figure 4.3**), because the single processing core is capable of executing only one thread at a time.

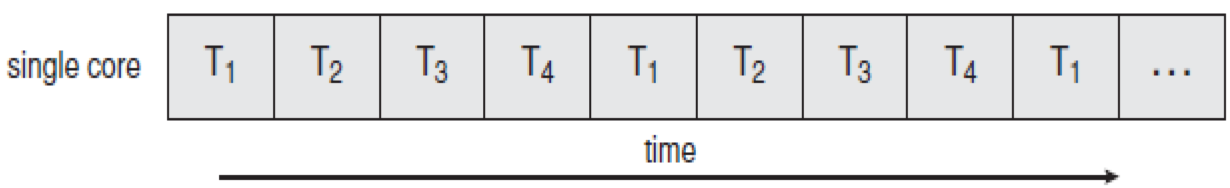


Figure 4.3 Concurrent execution on a single-core system.

Multi-core Programming

- On a system with **multiple cores**, *concurrency* means that the **threads** can run in **parallel**, because the system can assign a separate thread to each core (see Figure 4.4).

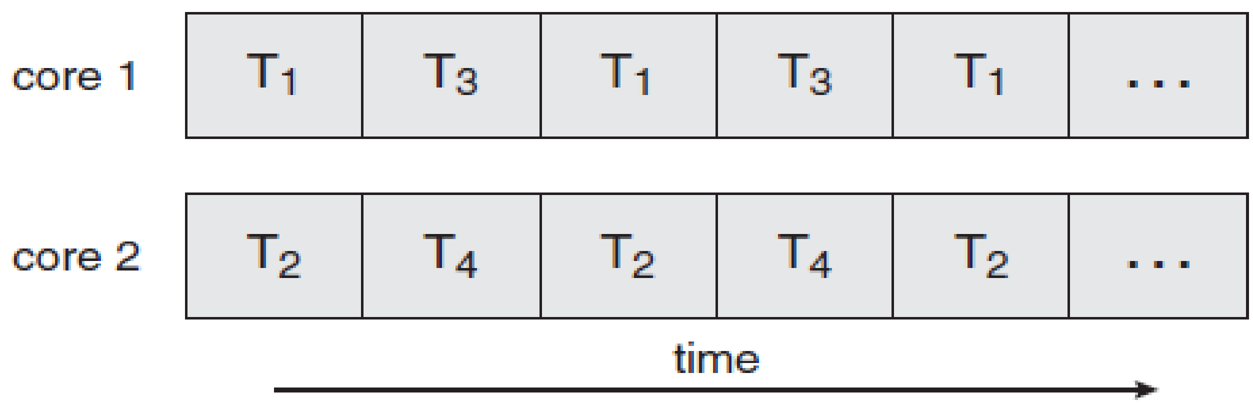


Figure 4.4 Parallel execution on a multicore system.

Parallelism and Concurrency

- **A system is parallel** if it can perform more than one task simultaneously.
- A **concurrent system** supports more than one task by allowing all the tasks to make progress.
  - **Thus, it is possible to have concurrency without parallelism.**
- In a **single core** system, the CPU schedulers were designed to provide the illusion of parallelism by rapidly switching between processes in the system, thereby allowing each process to make progress.
  - **Such processes were running concurrently, but not in parallel.**

Concurrent Processes (Cont.)

- Concurrent processes are often **non-deterministic**, which means it is not possible to tell, by looking at the process, what will happen when it executes
  - Consider two threads **A** and **B** and **a1** and **b1** are the events of **A** and **B** respectively:

Thread A: a1 print "yes" **Thread B: b1 print "no"**

- Because the two threads run **concurrently**, the order of execution depends on the scheduler
  - During any given run of this program, the output might be "yes no" or "no yes"

Concurrent Processes (Cont.)

- When **concurrent processes** (or **threads**) interact through a **shared variable**
  - may cause the integrity of the **variable** to be violated, if the variable access is not coordinated
  - Ex. Of integrity violations are:
    - The variable does not record all changes
    - A process may read inconsistent values
    - The final value of the variable may be inconsistent
- **Concurrent writes**
  - Consider, ***x*** is **a shared variable** accessed by two writes

Thread A		Thread B	
	a1: *x* = 5 **a2: print *x***		**b1: *x* = 7**

  - What value of ***x*** gets printed? What is the final value of ***x*** ?
  - What is the **execution path** of events of threads A and B?

Amdahl's Law

Amdahl's Law is a formula that identifies potential performance gains from adding additional computing cores to an application that has both serial (nonparallel) and parallel components. If S is the portion of the application that must be performed serially on a system with N processing cores, the formula appears as follows:

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

As an example, assume we have an application that is 75 percent parallel and 25 percent serial. If we run this application on a system with two processing cores, we can get a speedup of 1.6 times. If we add two additional cores (for a total of four), the speedup is 2.28 times.

One interesting fact about Amdahl's Law is that as N approaches infinity, the speedup converges to $1/S$. For example, if 40 percent of an application is performed serially, the maximum speedup is 2.5 times, regardless of the number of processing cores we add. This is the fundamental principle behind Amdahl's Law: the serial portion of an application can have a disproportionate effect on the performance we gain by adding additional computing cores.

Types of Parallelism

- In general, there are **two types** of parallelism:
  - **data parallelism** and
  - **task parallelism**.

Data parallelism

- **Data parallelism** focuses on *distributing the data across multiple cores* and performing the same operation on each core.
  - For example, summing the contents of an array of size ***N.***
  - On a **single-core system**, one thread would simply sum the elements [0] . . . [*N* − 1].
  - On a **dual-core system**, however, **thread *A**, running on **core0,** could sum the elements [0] . . . [*N*/2 − 1] while **thread B**, running on **core1** could sum the elements [*N*/2] . . . [*N* − 1].
    - These two threads would be running in parallel on separate computing cores.

Task parallelism

- **Task parallelism** involves distributing *tasks* (or threads) across **multiple cores**:
  - Each thread is performing a unique operation.
  - Different threads may be operating on the same data, or on different data.
  - An example of **task parallelism** might involve *two threads*, each performing a unique statistical operation on the array of elements.
  - The threads (tasks) again are operating in parallel on separate computing cores, but each is performing a unique operation.
- **Data parallelism** involves the distribution of data across multiple cores and **task parallelism** on the distribution of tasks across multiple cores.

Multithreading Models

- However, support for threads may be provided either at **the user level,** for user threads, or **by the kernel,** for kernel threads.
  - **User threads** are supported above the kernel and are managed without kernel support,
  - whereas **kernel threads** are supported and managed directly by the OS.
- Virtually all contemporary operating systems—including Windows, Linux, Mac OS X, and Solaris—support kernel threads.

Multithreading Models

- There is a relationship exists between **user threads** and **kernel threads**.
- There are **three ways** of mapping the **user threads** to **kernel thread** for thread execution:
  - **many-to-one** model
  - **one-to-one** model
  - **many-to-many** model

Many-to-one Model

- The **many-to-one** model (**Figure 4.5**) maps many **user threads** to **one kernel thread**.
- Thread management is done by the **thread library** in user space, so it is efficient.
- The entire process will block if a thread makes a **blocking system call**.
- only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems.
- For example, **Green threads**—a thread library available for Solaris systems and adopted in early versions of Java—used the many-to-one model.

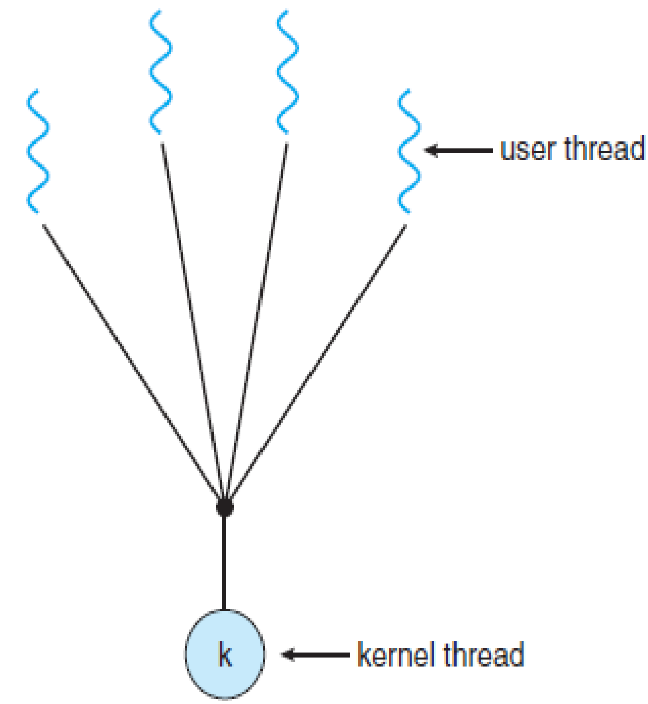


Figure 4.5 Many-to-one model.

One-to-one Model

- The **one-to-one model** (Figure 4.6) maps each **user thread** to individual **kernel thread**.
  - It provides more **concurrency** than the many-to-one model by allowing another thread to run when a thread makes a **blocking system call**.
  - It allows multiple threads to run in parallel on **multicore system** (greater concurrency).
  - The only **drawback** to this model is that creating a user thread requires creating the corresponding kernel thread:
    - because the overhead of creating **kernel threads** can burden the performance of an application.
  - **Linux**, along with the family of **Windows**, implement the **one-to-one** model.

One-to-One Model

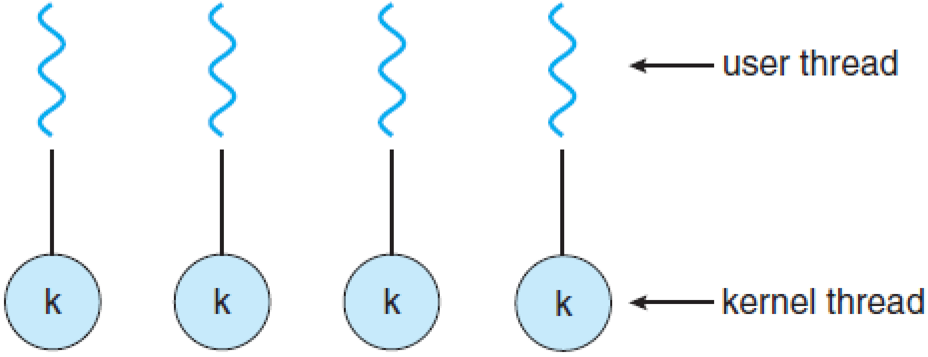


Figure 4.6 One-to-one model.

Many-to-Many Model

- The **many-to-many** model (**Figure 4.7**) multiplexes many **user threads** to a smaller or equal number of **kernel threads**.
  - The number of kernel threads may be specific to either a particular application (an application may be allocated more kernel threads on a multiprocessor than on a single processor).
  - The many-to-one model allows the developer to create as many user threads.
  - It does not result in true concurrency, because the kernel can schedule only one thread at a time.
  - Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.

Many-to-Many Model

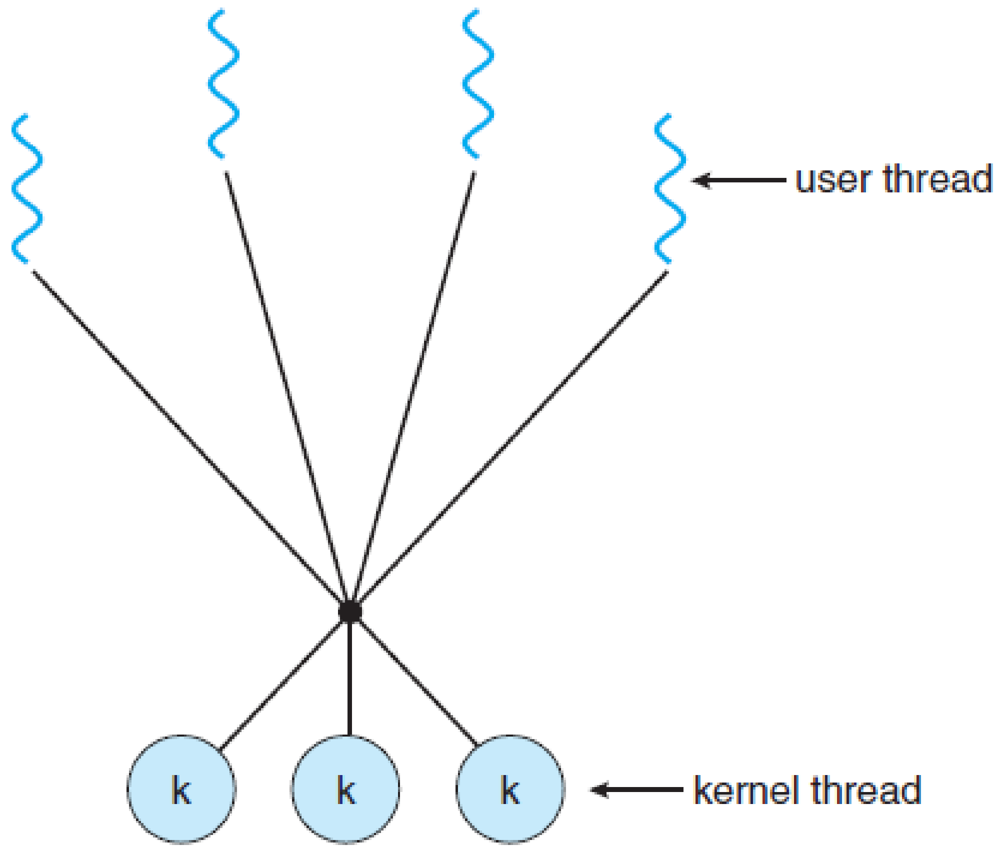


Figure 4.7 Many-to-many model.

Two-level Model

- One variation on the **many-to-many** model still multiplexes many **user level threads** to a smaller or equal number of **kernel threads** but also allows a **user-level thread** to be bound to a **kernel thread**.
- This variation is sometimes referred to as the **two-level model** (**Figure 4.8**). The **Solaris** supported the two-level model in versions older than Solaris 9.
- However, beginning with Solaris 9, this system uses the **one-to-one model.**

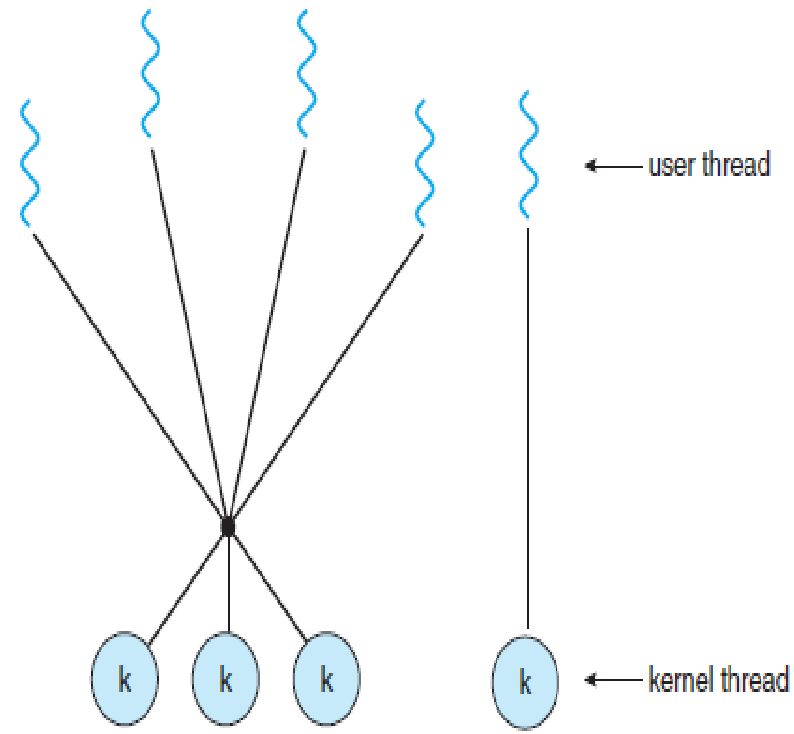


Figure 4.8 Two-level model.

Java Threads

- Threads are the fundamental model of program execution in a **Java** program.
- All Java programs comprise at least a single thread of control—even a simple Java program consisting of only a **main() method** runs as a single thread in the **JVM**.
- Java threads are available on any system that provides a JVM including Windows, Linux, and Mac OS X.
- The Java thread API is available for Android applications.
- There are two techniques for creating threads in a Java program.
  - One approach is to create a new class that is derived from the Thread class and to override its **run()** method.
  - A more commonly used—technique is to define a class that implements the **Runnable** interface.

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:

```
public interface Runnable
{
    public abstract void run();
}
```

  - Extending Thread class
  - Implementing the Runnable interface

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Signal handling
  - **Synchronous and asynchronous**
- Thread cancellation of target thread
  - **Asynchronous or deferred (delayed)**
- Thread-local storage
- Scheduler Activations

Thread Scheduling

- We have seen the distinction between **user-level** and **kernel-level** threads
- As per operating systems **the kernel-level threads**—not processes—that are being scheduled by the operating system.
- **User-level threads** are managed by a **thread library**, and the kernel is unaware of them. To run on a CPU, **user-level threads** must ultimately be mapped to an associated **kernel-level thread**, although this mapping may be indirect and may use a **lightweight process (LWP)**.
- **Thread library schedules user-level threads to run on LWP**
  - Known as **process-contention scope (PCS)** since scheduling competition is within the process
  - Typically done via priority set by programmer
- **Kernel thread** scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system

Thread Scheduling

- When we say the ***thread library*** schedules **user threads** onto available **LWP**s, we do not mean that the threads are actually running on a CPU.
- That would require the OS to schedule the **kernel thread** onto a physical CPU. To decide which **kernel-level thread** to schedule onto a CPU, the kernel uses **system-contention scope (SCS).**
- Typically, **process-contention scope (PCS)** is done according to priority—the scheduler selects the runnable thread with the highest priority to run.
- **User-level thread** priorities are set by the programmer and are not adjusted by the **thread library**, although some thread libraries may allow the programmer to change the priority of a thread.
- It is important to note that **PCS** will typically preempt the thread currently running in favor of a **higher-priority thread**.

Semantics of fork() and exec()

- Does `fork()` duplicate only the calling thread or all threads?
  - Some UNIXs have two versions of fork
- `exec()` usually works as normal – replace the running process including all 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
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled by one of two signal handlers:
     1. default
     2. user-defined
- Every signal has **default handler** that kernel runs when handling signal
  - **User-defined signal handler** can override default
  - For single-threaded, signal delivered to process

Signal Handling (Cont.)

- 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

Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is called a **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
- **Pthread** code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

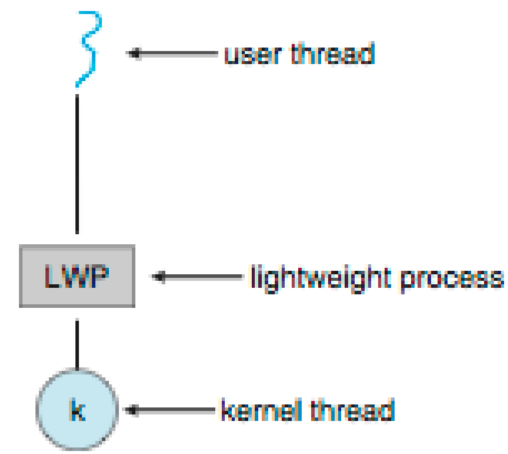
- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches **cancellation point**
    - I.e. `pthread_testcancel()`
    - Then **cleanup handler** is invoked
- On **Linux** systems, thread cancellation is handled through signals

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

Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an **intermediate data structure** between **user** and **kernel** threads – **lightweight process** (**LWP**)
  - **Appears to be a virtual processor on which process can schedule user thread to run**
  - **Each LWP attached to kernel thread**
  - **How many LWPs to create?**
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the **thread library**
- This communication allows an application to maintain the **correct number kernel threads**



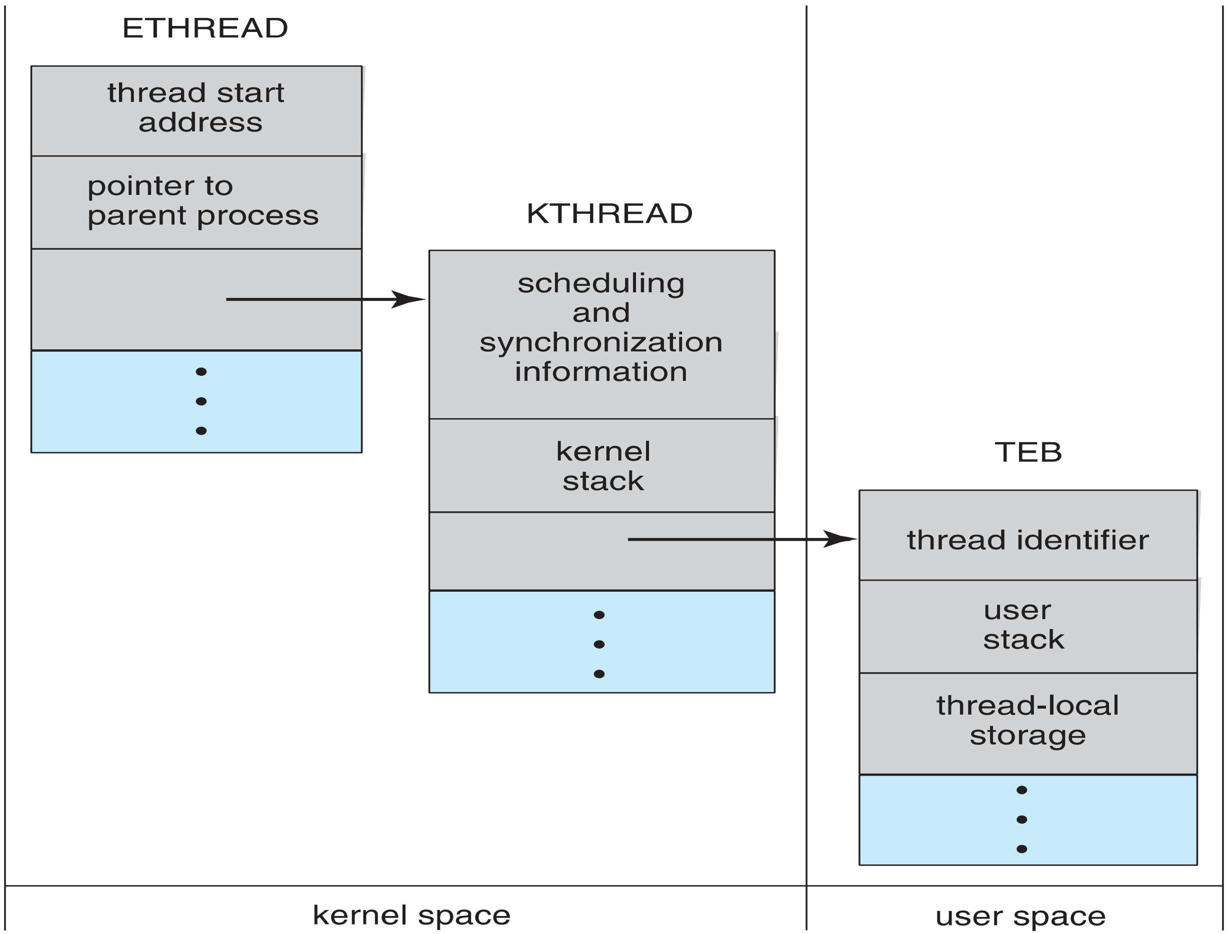
Windows Threads

- Windows implements the Windows API – primary API for Win 98, Win NT, Win 2000, Win XP, and Win 7
- Implements the **one-to-one mapping**
- **Each thread contains**
  - A thread id
  - Register set representing state of processor
  - Separate user and kernel stacks for when thread runs in **user mode** or **kernel mode**
  - Private data storage area used by run-time libraries and **dynamic link libraries** (**DLL**s)
- The register set, stacks, and private storage area are known as the **context** of the thread

Windows Threads (Cont.)

- The primary data structures of a **thread** include:
  - **ETHREAD** (Executive Thread Block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
  - **KTHREAD** (Kernel Thread Block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
  - **TEB** (Thread Environment Block) – thread id, user-mode stack, thread-local storage, in user space

Windows Threads Data Structures



Linux Threads

- **Linux** refers to them as ***tasks*** rather than ***threads***
- **Thread creation** is done through **`clone()`** system call
- **`clone()`** allows a child task to share the address space of the parent task (process)
  - Flags control behavior

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.

- **`struct task_struct`** points to process data structures (shared or unique)

End of Chapter 4