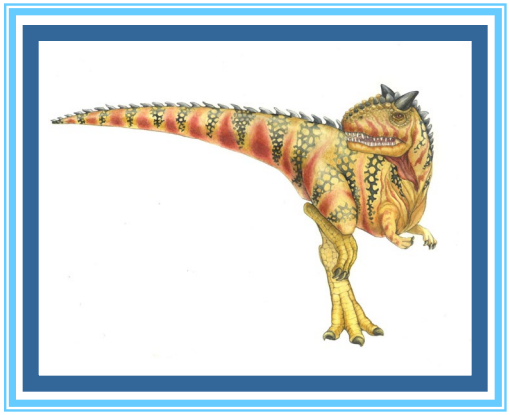
Chapter 4: Threads



Introduction

- A single CPU process model introduced in the previous topic assumed that a ***process was executing a program with a single thread of control***.
- All modern operating systems provide features enabling a process to contain **multiple threads of control**.
  - Identifying opportunities for parallelism through **thread-based parallelism** is important for modern **multicore operating systems**.
- This topic aims to introduce concepts and design challenges associated with **multithreaded operating systems**, including the *APIs for the Pthreads*, *Windows*, and *Java thread libraries*.
  - Several new OS features that abstract the *creation of threads, allowing developers to focus on identifying opportunities for thread parallelism* while letting language features handle the user thread creation and management.

Overview-Thread

- A **thread** is a basic unit of CPU utilization (in a multi-core environment);
  - It comprises a ***thread ID***, a ***program counter (PC)***, a ***register set***, and a ***stack***.
  - It shares with other threads in the same process its code and data sections, as well as other operating-system resources, such as open files and signals.
- A **traditional process has a single control thread** (as per a single-core system).
  - 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.

Overview-Thread

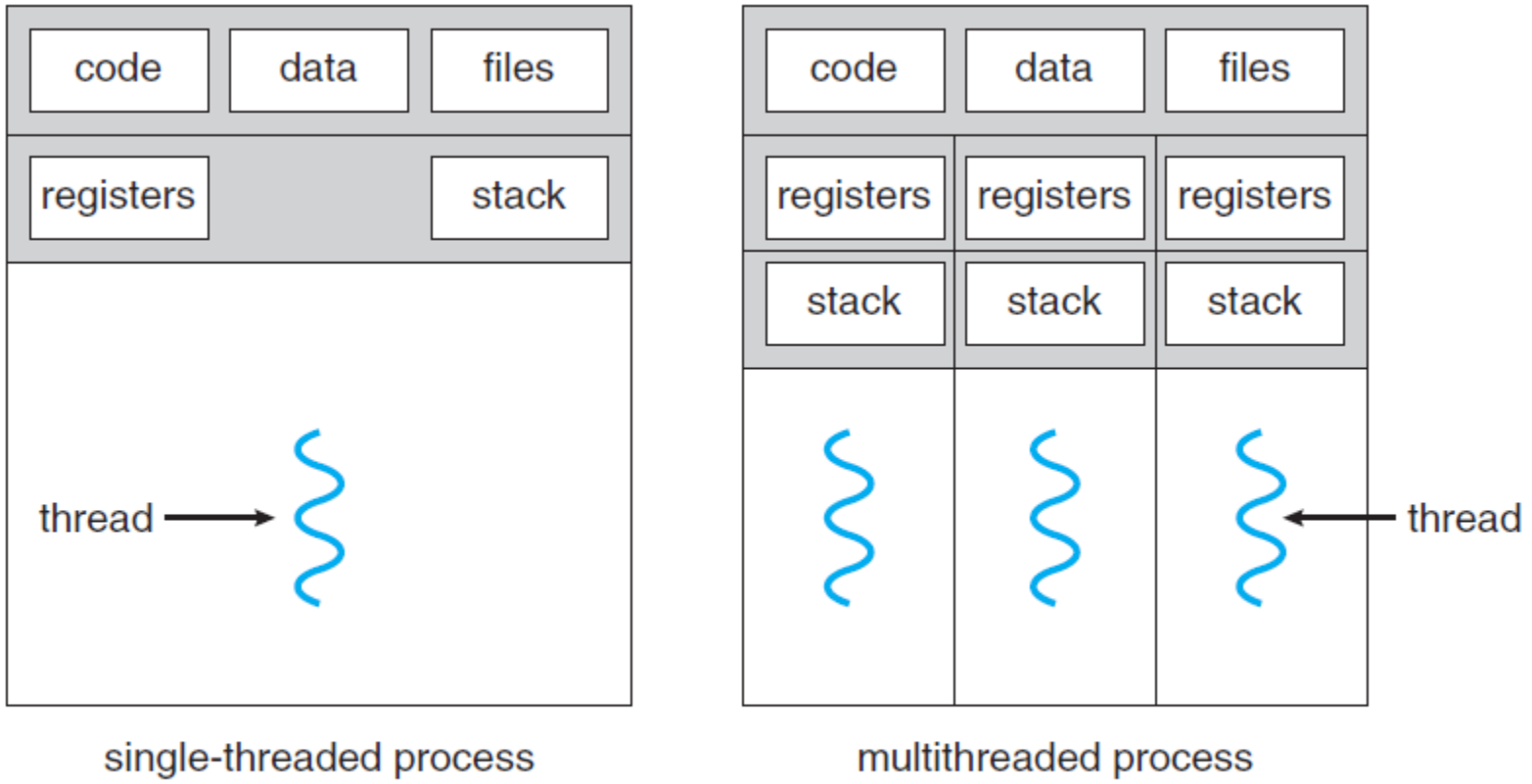


Figure 4.1 Single-threaded and multithreaded processes.

Motivation

- Most software applications that run on modern computers and mobile devices are **multithreaded**.
  - An application typically is implemented as a **separate process with several threads of control**.
  - Below **highlight a few examples of multithreaded applications**:
    - *A web browser might have one thread display images or text while another thread retrieves data from the network*.
    - *A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.*

Motivation

- In certain situations, **a single application** may be required to perform **several tasks**.
  - For example, **a web server** accepts client requests for web pages, images, sound, and so forth.
    - ***A busy web server may have several clients accessing it concurrently***.
  - If the **web server** ran as a traditional **single-threaded process**, it could service only **one client at a time**, and a **client** might have to wait a very long time for its request to be processed.
    - Here, the **server runs as a single process** that accepts **requests**.
    - When the **server receives a request**, it creates a separate process to handle it.

Motivation

- **Process creation is time-consuming and resource-intensive.**
- If the **new process** will perform the same tasks as the **existing process**, *it incurs overhead*.
  - 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 resumes listening for additional requests*.
  - This is illustrated in **Figure 4.2**.
    - **Where 1, 2, and 3 are threads of the web-server process**

Motivation

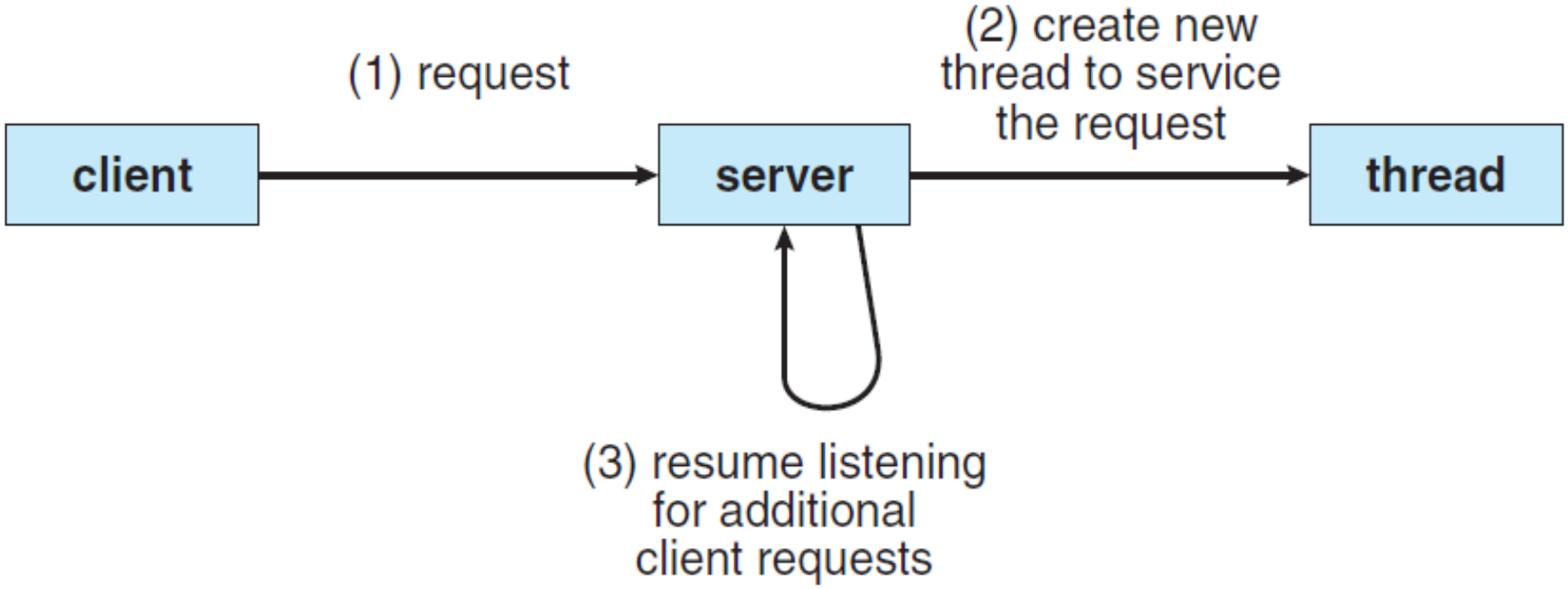


Figure 4.2 Multithreaded server architecture.

Motivation

- Most **operating system kernels** are also typically **multithreaded**.
  - As an example, during ***system boot time*** on **Linux systems**, ***several kernel threads are created***.
  - Each thread performs a specific task, such as *managing devices, memory management*, or *interrupt handling*.
    - The command **`ps -ef`** can be used to ***display the kernel threads*** on a running **Linux system**.
    - Examining the **output of this command** shows the kernel thread **`kthreadd`** (**`pid = 2`**), which serves as the parent of all other **kernel threads**.

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 sharing code and data 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 of how many are available.

- Modern computer systems are designed with **multiple computing cores or CPU cores on a single chip**, where each core appears as a separate CPU to the operating system.
  - Such systems are referred to as **multicore**, and ***multithreaded programming*** provides a *mechanism for more efficient use of these multiple computing cores and improved execution concurrency.*

Multi-core Programming

- **Multiple computing cores** are designed in a single-chip form (we call these **multi-core** or **multiprocessor** systems)
- Each core appears as a **separate processor** to the OS.
  - **Multithreaded programming** provides a mechanism for more efficient use of these **multiple cores**.
  - Consider an application with **four threads** on a **single-core CPU**. 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***.

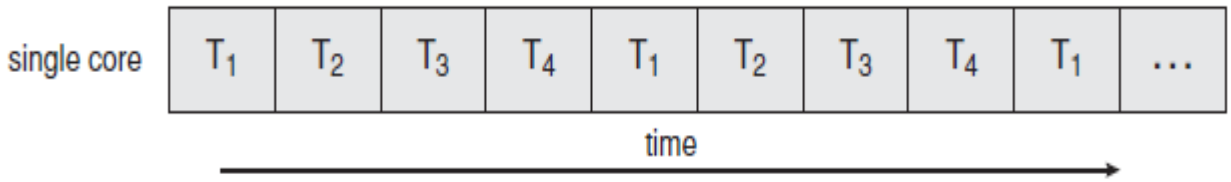


Figure 4.3 Concurrent execution on a single-core system.

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).

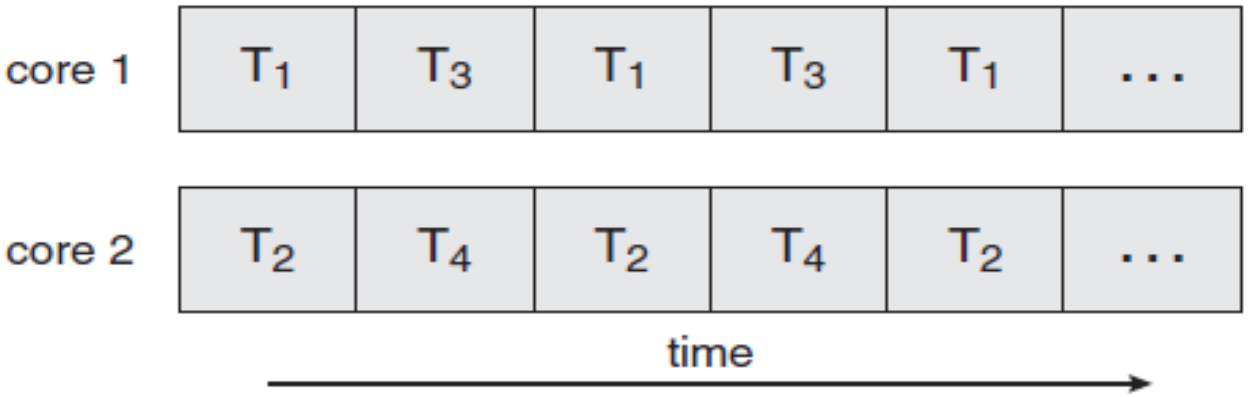
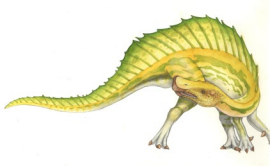


Figure 4.4 Parallel execution on a multicore system.

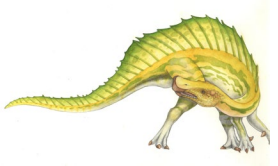
Parallelism and Concurrency

- A **concurrent system** supports more than one task by allowing all the tasks to make progress.
- A **parallel system** can perform multiple tasks simultaneously.
- **Thus, it is possible to have concurrency without parallelism**.
  - Before the **multiprocessor and multicore architectures**, most computer systems had only a **single processor**, and **CPU schedulers** were designed to provide the illusion of parallelism by rapidly switching between processes, thereby allowing each process to make progress.
  - **Such processes were running concurrently, but not in parallel**.



Data and Task 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.



Data and 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** are again **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** involves the distribution of tasks across **multiple cores**, as shown in **Figure 4.5**.

Data and Task parallelism

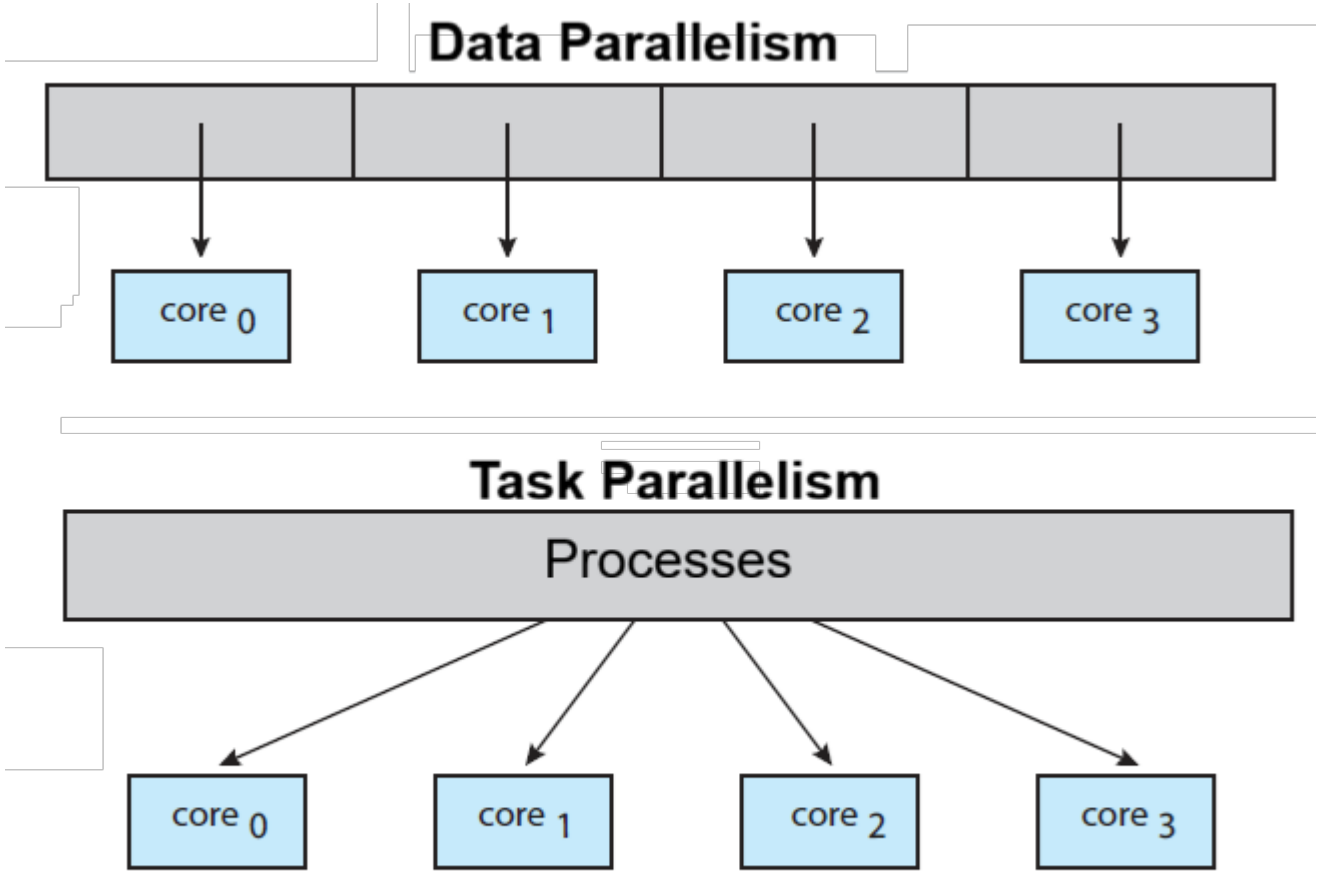


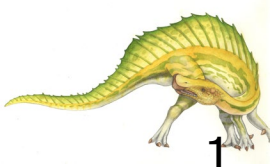
Figure 4.5 Data and task parallelism.

Parallelism and Concurrency

- 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"

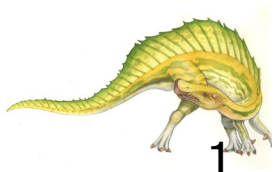


Concurrent Processes (Cont.)

- 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	**a1: *x* = 5** **a2: print *x***	**Thread B**	**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.

Multithreading Models

- OS support for thread execution may be either at the **user level**, for **user threads**, or at the **kernel level**, for **kernel threads**.
  - **User threads** are supported above the **OS kernel** and are managed without kernel support.
  - whereas **kernel threads** are supported and managed directly by the **operating system**.
- Virtually all contemporary operating systems—including **Windows**, **Linux**, and **macOS**— support the generation of **kernel threads**.
  - A relationship that exists between **user threads and kernel threads**, as illustrated in **Figure 4.6**.
  - Three common ways of establishing such a relationship: the **many-to-one model**, the **one-to-one model**, and the **many-to-many model**.

Multithreading Models

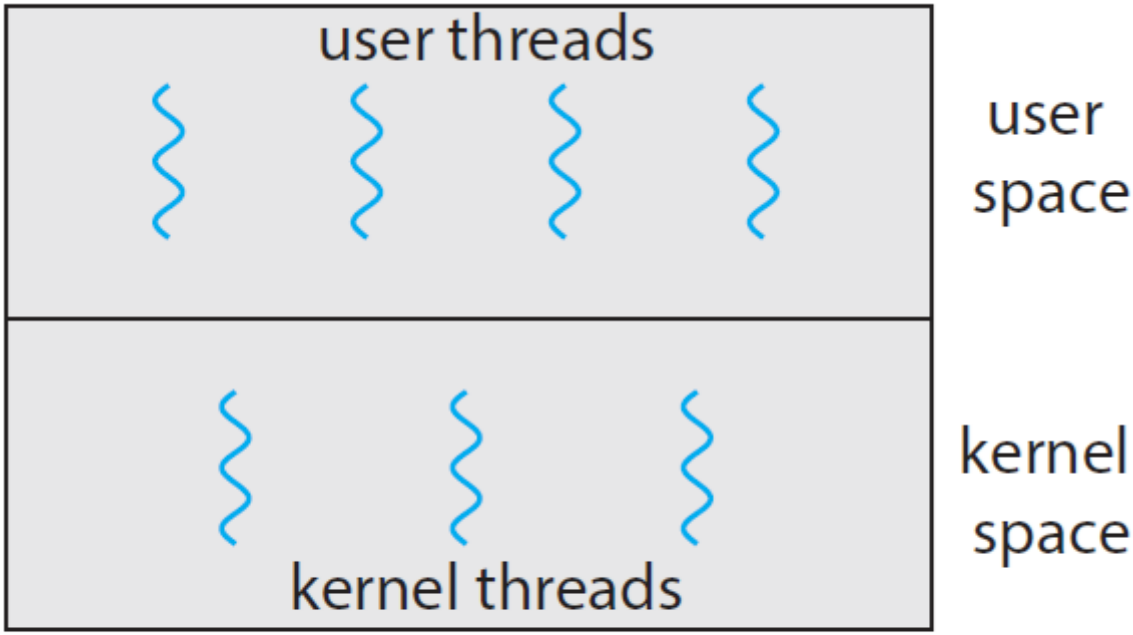


Figure 4.6 User and kernel threads.

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 4.7**) maps many **user threads** to **one kernel thread**.
- The thread library does thread management 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.

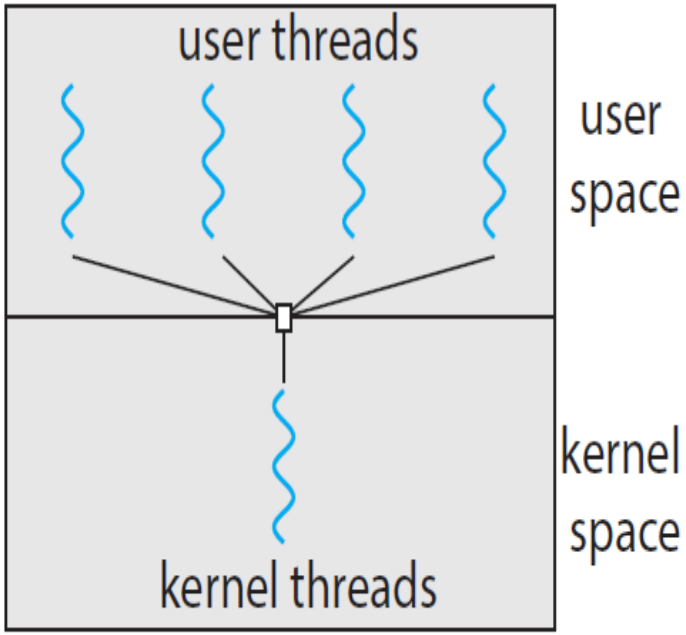


Figure 4.7 Many-to-one model.

One-to-one Model

- The **one-to-one model** (Figure 4.8) maps each **user thread** to an 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 a **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**, implements the **one-to-one** model.

One-to-One Model

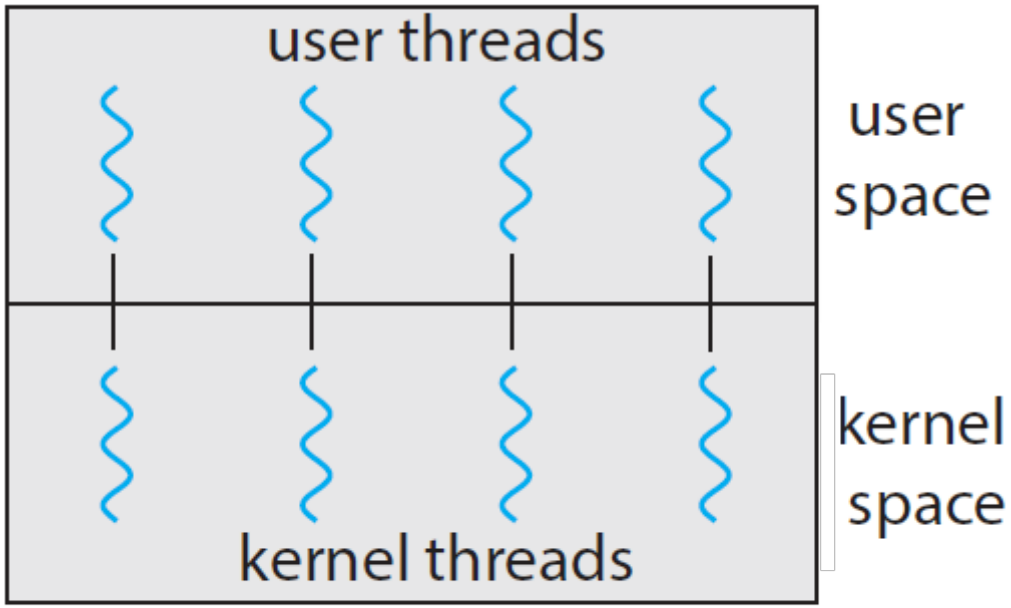


Figure 4.8 One-to-one model.

Many-to-Many Model

- The **many-to-many** model (**Figure 4.9**) 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

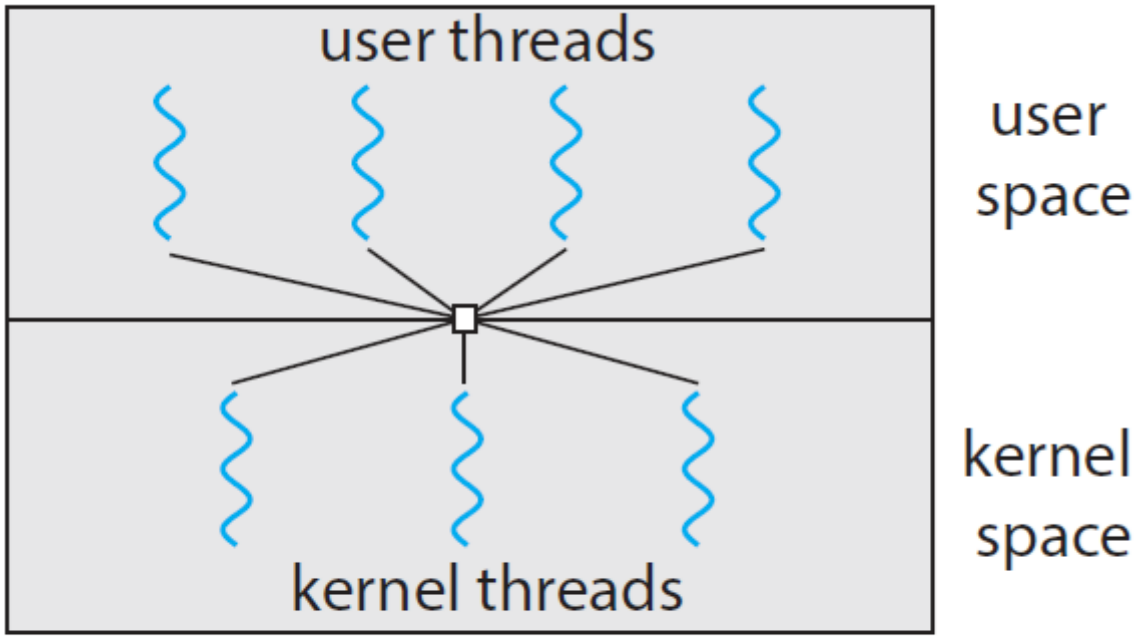


Figure 4.9 Many-to-many model.

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 4.10**).
- **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.**

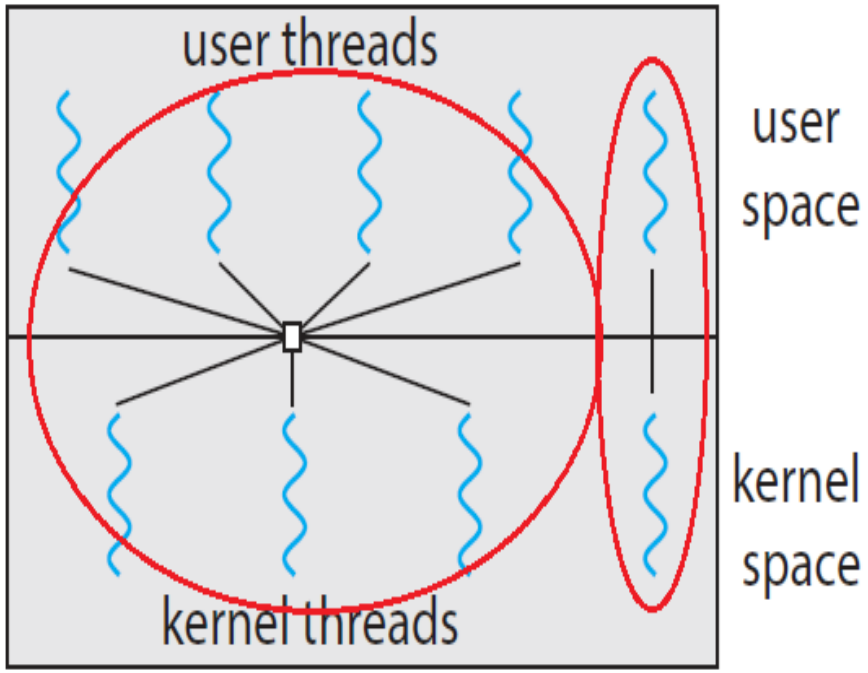


Figure 4.10 Two-level model.

Thread Libraries

- A **thread library** provides the programmer with an **API for creating and managing threads**. There are **two primary ways of implementing a thread library**.
  - **The first approach:**
    - is to provide a library entirely in **user space with no kernel support.**
    - *All code and data structures for the library exist in user space.*
    - This means that invoking a function in the library results in a local function call in user space and **not a system call**.
  - **The second approach:**
    - is to implement a kernel-level library supported directly by the **operating system**.
    - In this case, code and data structures for the library exist in kernel space. Invoking a function in **the library's API** typically results in a system call to the kernel.

Java Threads

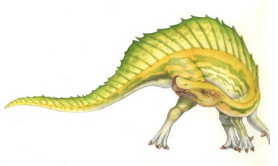
- Threads are the fundamental model of program execution in a **Java** program.
- All Java programs comprise at least a single thread of control—even a simple Java program consisting of only a **main() method** runs as a single thread in the **JVM**.
- Java threads are available on any system that provides 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



Scheduler Activations

- Many systems implementing either the **many-to-many** or the **two-level model** place an **intermediate data structure** between the **user and kernel threads**.
- This data structure—typically known as a **lightweight process**, or **LWP**—is shown in **Figure 4.20**.

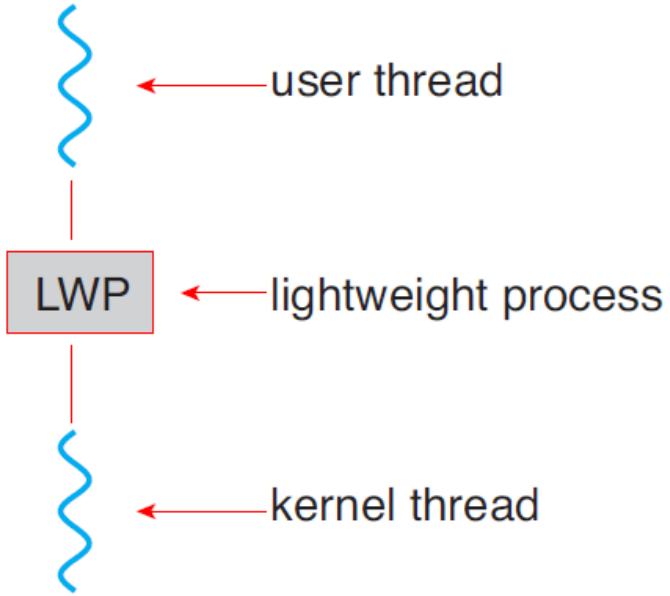


Figure 4.20 Lightweight process (LWP).

Scheduler Activations

- To the **user-thread library**, the **LWP** appears to be a **virtual processor** on which the application can schedule a **user thread** to run.
  - Each **LWP** is attached to a **kernel thread**, and it is kernel threads that the OS schedules to run on physical processors.
- If a **kernel thread blocks** (such as *while waiting for an I/O operation to complete*), the **LWP blocks** as well.
- The **user-level thread attached to the LWP also blocks**.
- Typically, an LWP is required for each **concurrent blocking system call**. Suppose, for example, that **five different file-read requests** occur simultaneously.
  - **Five LWPs** are needed because all could be waiting for I/O completion in the kernel.
  - If a process has only **four LWPs**, then the **fifth request must wait for one of the LWPs to return from the kernel**.

Scheduler Activations

- ***One scheme for communication between the user-thread library and the kernel*** is known as **scheduler activation**.
- **It works as follows:**
  - The kernel provides an application with a set of **virtual processors (LWPs)**, and the application can schedule **user threads** onto an available **virtual processor**.
  - The **kernel** must inform an application about certain events.
  - This procedure is known as an **upcall**.
  - The thread library handles upcalls with an **upcall handler**, and **upcall handlers** must run on a **virtual processor**.

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:

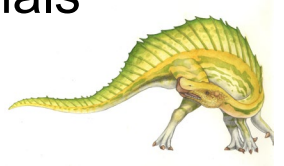
```
pthread_t tid;

/* 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

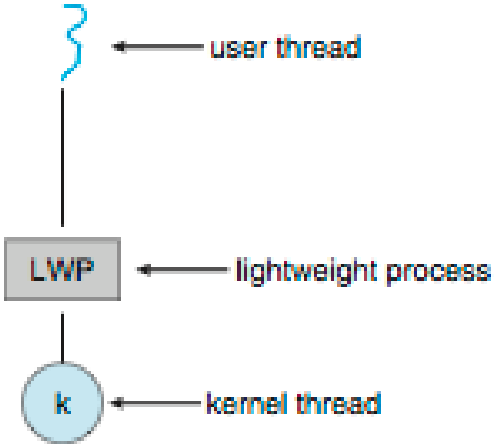


Thread-Local Storage

- **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**



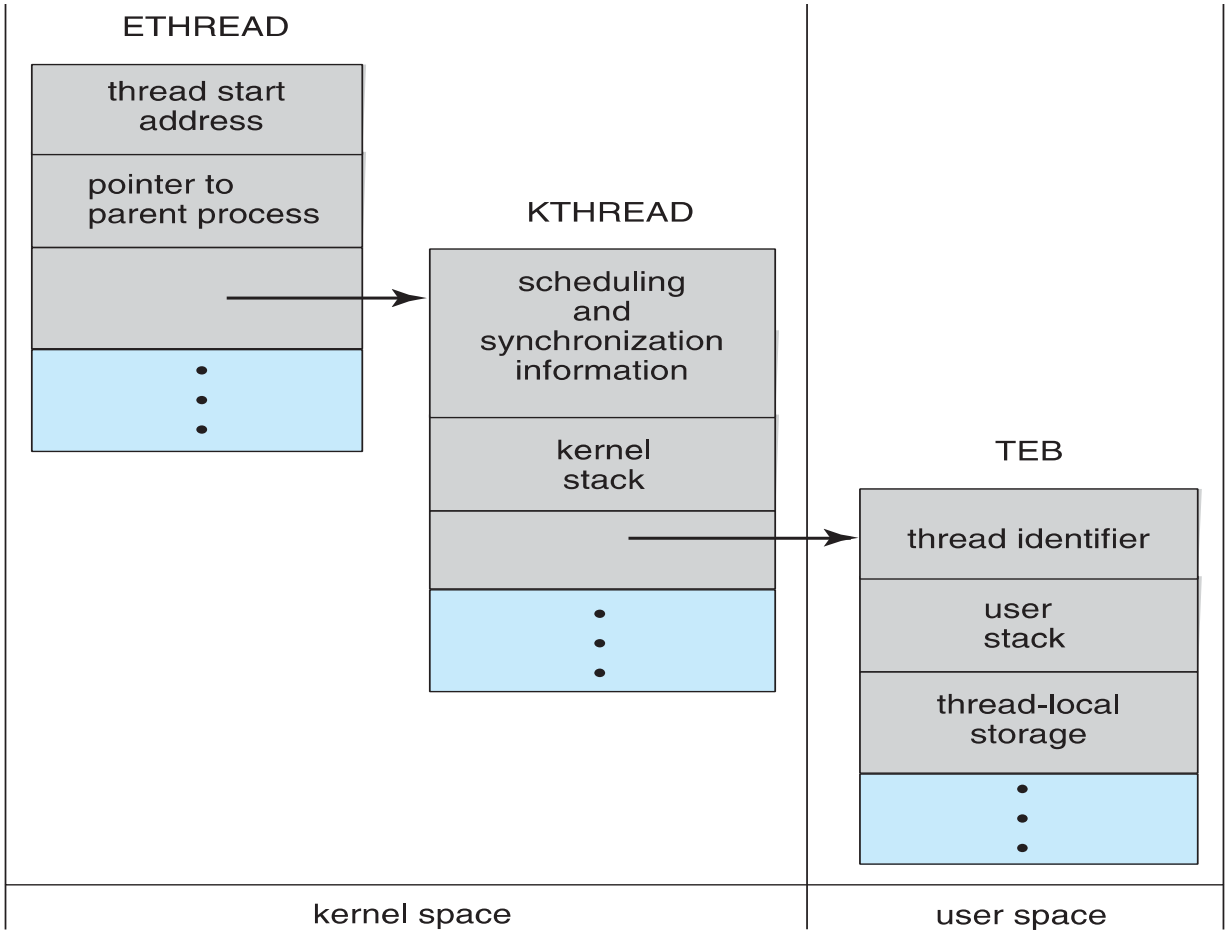
Windows Threads

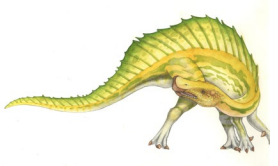
- Windows implements the Windows API – primary API for Win 98, Win NT, Win 2000, Win XP, and Win 7
- Implements the **one-to-one mapping**
- **Each thread contains**
  - A thread id
  - Register set representing state of processor
  - Separate user and kernel stacks for when thread runs in **user mode** 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)