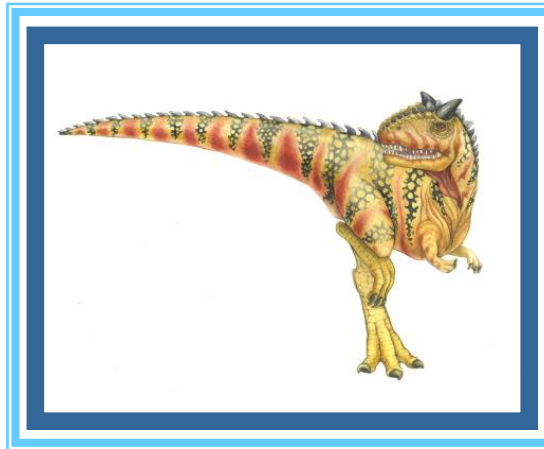


线程与并发性

# Chapter 4: Threads & Concurrency

---





# Chapter 4: Threads

- Overview
- Multicore Programming
- Multithreading Models
- Threading Issues
- Operating System Examples

当然可以。以下是一个具体的例子来说明进程和线程的区别：

假设你在使用一个文本编辑器（如 Microsoft Word）：

## 1. 进程：

- 当你打开 Microsoft Word 时，操作系统会为这个程序创建一个进程。这个进程包含了 Word 的所有代码、数据和资源（如打开的文件、字体、样式等）。
- 如果你同时打开了多个程序，比如浏览器、音乐播放器和文本编辑器，每个程序都会有自己的进程，它们之间是相互独立的。

## 2. 线程：

- 在 Microsoft Word 进程内部，可能会有多个线程在运行。例如，一个线程负责处理用户输入（如键盘和鼠标操作），另一个线程可能在后台自动保存文档，第三个线程可能负责拼写检查。
- 这些线程共享 Word 进程的资源，比如打开的文档和设置，但每个线程有自己的执行路径和局部变量。

通过这个例子，可以看到进程是一个独立的程序实例，而线程是在这个程序内部执行的多个任务。





# Objectives

---

- ❑ Identify the basic components of a thread, and contrast thread and processes.
- ❑ Describe the major benefits and challenges in designing multithreaded processes.
- ❑ Describe how Windows and Linux operating systems represent threads





# Motivation

- ❑ Many applications or programs are **multithreaded**
  - ❑ A web browser might have one thread displaying images or text while another thread retrieving data from the network
  - ❑ A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from user, and a third thread for performing spelling and grammar checking in the background
- ❑ Most operating system kernels are multithreaded
  - ❑ For example, during system boot time on Linux, several kernel threads are created. Each thread performs a specific task, such as managing devices, memory management, or interrupt handling
- ❑ It can take advantage of processing capabilities on multicore systems
  - ❑ Parallel programming widely used in applications such as data mining, graphics, and artificial intelligence
- ❑ Process creation is time consuming and resource intensive
  - ❑ **Process creation** is **heavy-weight** while **thread creation** is **light-weight**, in which different threads belonging to the same process share code, data and others





# Examples of Multithreaded Programs

---

- ❑ Embedded systems
  - ❑ Elevators, planes, medical systems, wristwatches
  - ❑ Single program, concurrent operations
- ❑ Most modern OS kernels
  - ❑ Internally concurrent to deal with concurrent requests by multiple users
  - ❑ But no protection needed within kernel
- ❑ Database Servers
  - ❑ Access to shared data by many concurrent users
  - ❑ Also background utility processing must be done
- ❑ Network Servers
  - ❑ Concurrent requests from network
  - ❑ Again, single program, multiple concurrent operations
  - ❑ File server, Web server, and airline reservation systems
- ❑ Parallel Programming
  - ❑ Split a program and data into multiple threads for parallelism

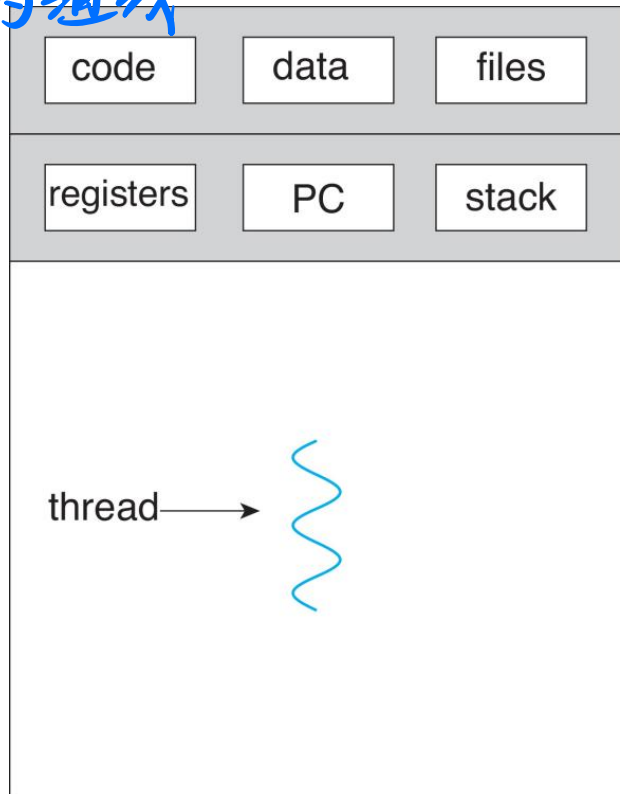




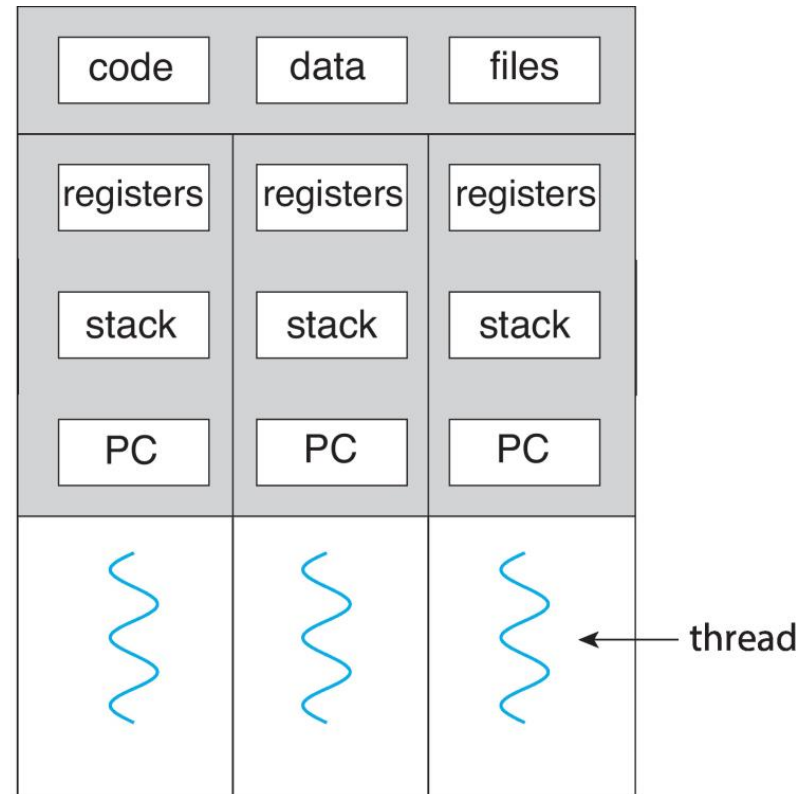
# Single and Multithreaded Processes

- Recall that a **thread** is a basic unit of CPU utilization – independently scheduled and run (referred as an **instance of execution**) represented by a **thread ID, a program counter (PC), a register set, and a stack**. It shares with other threads of the same process its code, data, and other OS resources, such as open files and signals

线程的组成

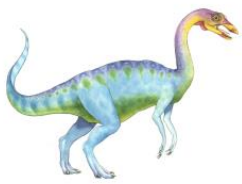


single-threaded process



multithreaded process





# Benefits

响应性

- ❑ **Responsiveness** – may allow continued execution if part of process (e.g., one thread of a process) is blocked, this could be especially important for interactive applications having user interfaces
- ❑ **Resource Sharing** – threads share resources of a process by default, easier than shared memory or message passing between processes – as they essentially run within the same address space of the process
- ❑ **Economy** – thread creation is much “cheaper” than process creation (consuming less time and memory) , context switching is typically faster between threads of a process than between processes
- ❑ **Scalability** – processes can take advantage of multicore architecture

可扩展性





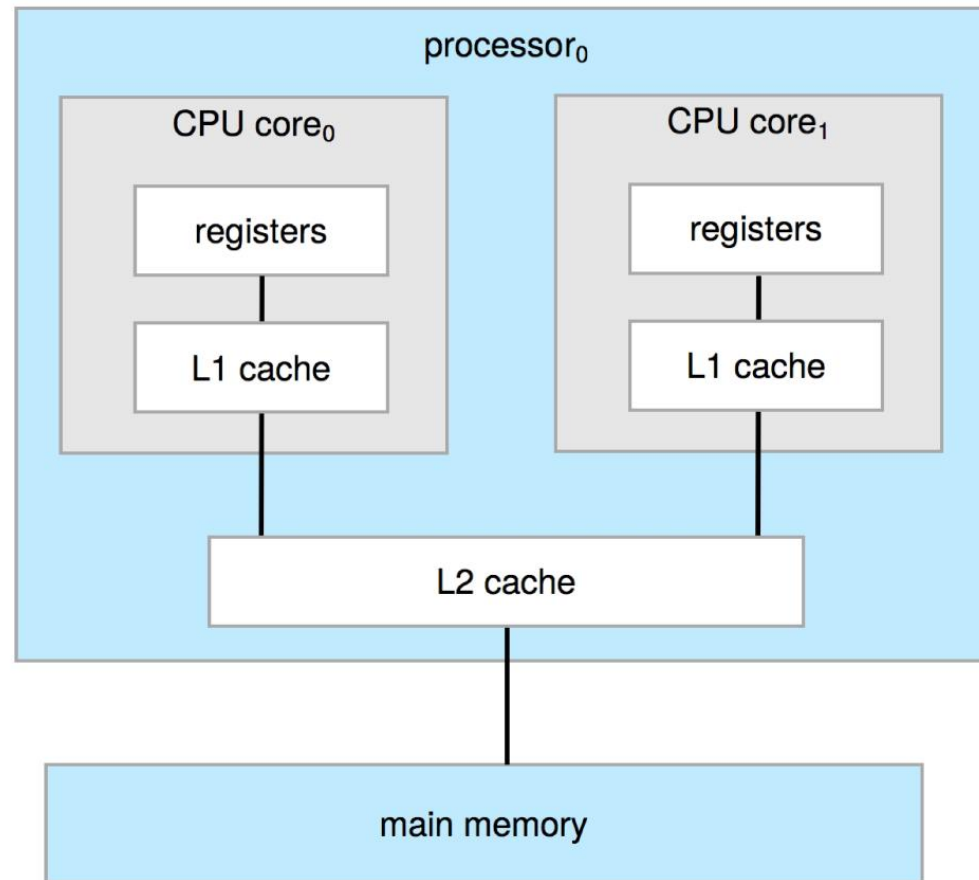
# A Multi-Core Design

多核设计

- The multithreaded programming provides a mechanism for more efficient use of multicores and improves concurrency in **multicore** systems.

机制

并发性







# Multicore Programming

- In **multicore** or **multiprocessor** systems, there are significant and new **challenges** in programming design include:
  - Dividing tasks – how to divide into separate, concurrent tasks
  - Balance – each task perform “equal” amount of work
  - <sup>数据拆分</sup> Data splitting - data used by tasks must be divided to run on separate cores
  - <sup>数据依赖</sup> Data dependency – if there is a data dependency, <sup>同步</sup> synchronization is required
  - Testing and debugging – different path of executions makes debugging difficult
- There is a clear distinction <sup>区别</sup> between **parallelism** <sup>并行性</sup> and **concurrency** <sup>并发性</sup>
  - **Parallelism** - a system can perform more than one task **simultaneously**
  - **Concurrency** supports more than one task for making progress
    - Multiplexed over time, single processor core, scheduler provides concurrency
- The advent of multicore systems require an entirely new approach in designing software systems with particular emphasis on parallel programming.

**并行性:** 强调能同时执行多个任务

**并发性:** 强调在同一段时间内处理多个任务的能力。(它能在多个任务之间快速切换,但在任何时刻他手上只能做着-件事)



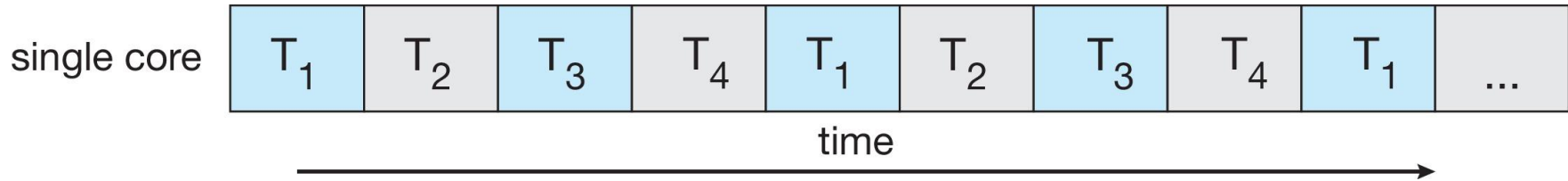


# Concurrency vs. Parallelism

并发执行

单核处理器

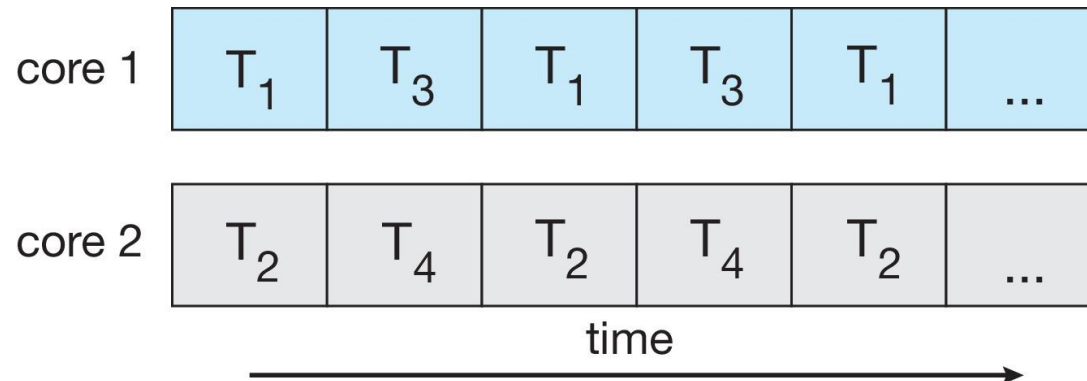
- Concurrent execution on single-core system – multiplexing over time



并行

多核

- Parallel execution on a multi-core system:



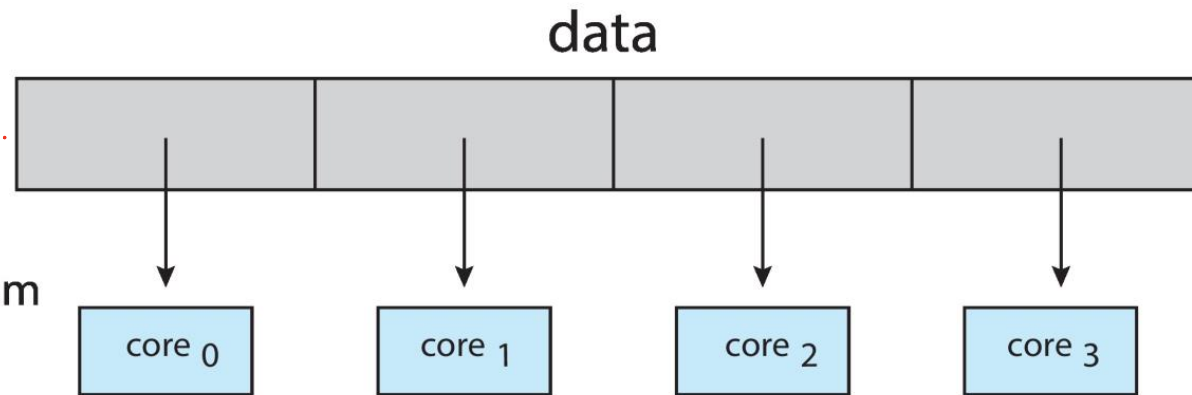


# Data and Task Parallelism

- **Data parallelism** – distributes subsets of the data across multiple cores, same operation on each core (common in distributed machine learning tasks)
- **Task parallelism** – distributing threads across cores, each thread performing unique operation
- Data and task parallelism are not mutually exclusive, an application may use both - hybrid

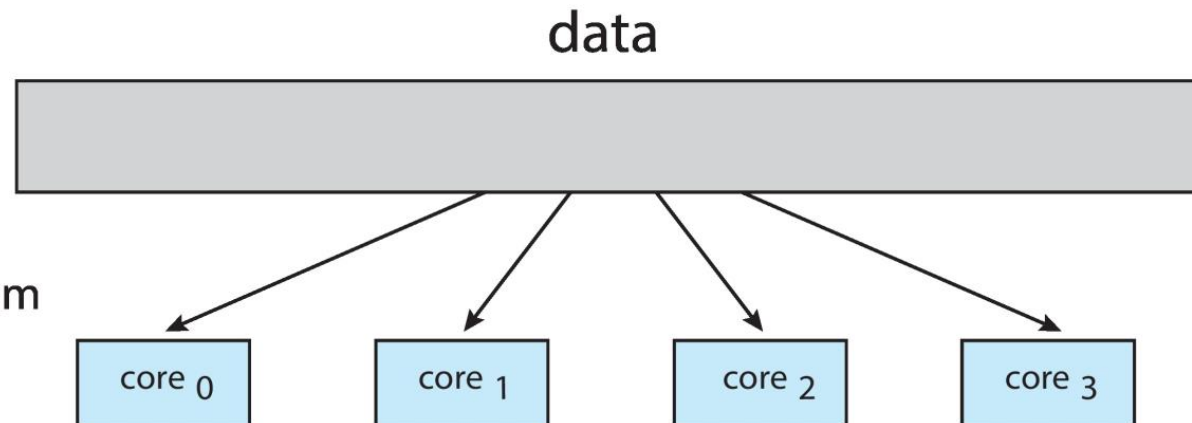
执行相同的操作.

data  
parallelism



执行不同的任务.  
然后合并.

task  
parallelism





# Amdahl's Law 阿姆达尔定理

- 识别 性能 提升  
□ It identifies performance gains (theoretical speedup in latency of the execution of a task at fixed workload) from adding additional cores to an application that has both serial and parallel components (in program)
- 串行部分 并行部分  
□  $S$  is serial portion and  $1-S$  is parallel portion
- $N$  processing cores

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

系统的性能提升受到串行部分的限制

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

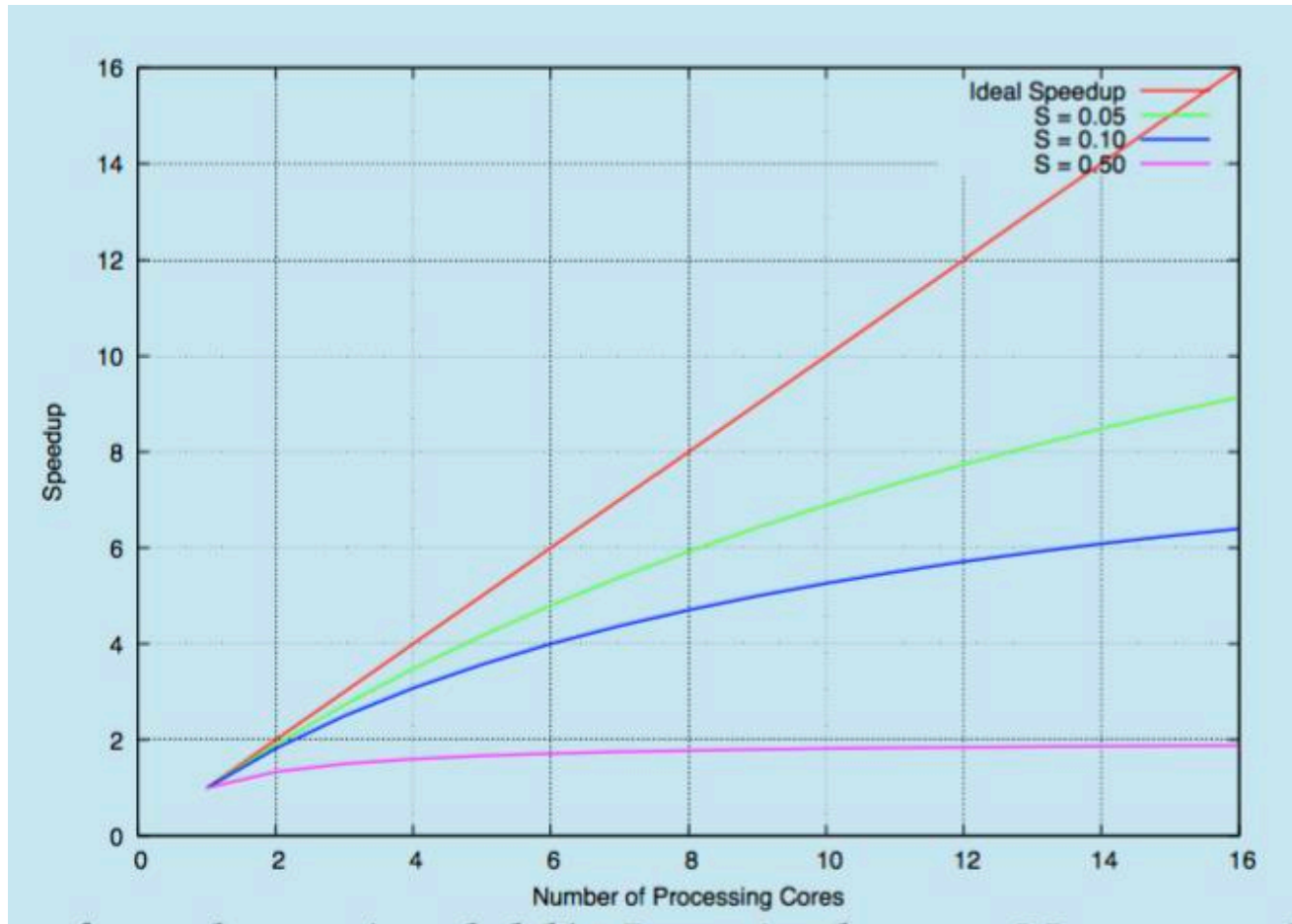
不成比例的影响

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





# Amdahl's Law

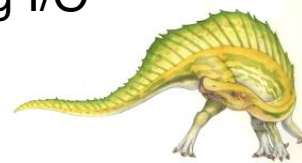


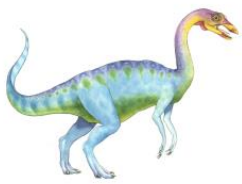


# Multithreaded Process

执行实例

- A multi-threaded process has more than one **instance of execution** - each of which with a **program counter** is being fetched and executed
- A thread is similar to a process, except they share the same address space and thus can access the same set of data with other thread(s) in a process
- The **state** of a single thread is also similar to that of a process - It has a **program counter (PC)** that tracks where the program is fetching instructions from. **Each thread has its own private set of registers for execution**
- If two threads run on a single processor, when switching from running one (T1) to running the other (T2), **a context switch must also take place**
  - The register state of T1 must be saved and the register state of T2 restored from T2's stack before running T2
  - The address space remains the same, the context switch overhead is much smaller
  - When switching to a thread belonging to a different process, overhead is more
- This provides **parallelism** in a process (multithreaded) execution and can enable **overlap** of I/O with other activities within a single program
  - One thread is running on CPU, and another thread of the process is doing I/O





# Thread

指的是每个线程在运行时  
自己独立的环境

- **Thread**: The **single unique** execution context - **lightweight process**
  - **Program counter, registers, execution flags, stack**
  - A thread is executing on a processor when it is resident in the registers.
  - PC register holds the address of executing instruction in the thread
  - Registers hold the root state of the thread (other state in memory)
- Each thread has a **Thread Control Block (TCB)** **包含与线程相关的所有信息**
  - **Execution state**: CPU registers, program counter, pointer to stack
  - **Scheduling info**: state (more later), priority, CPU time
  - **Accounting Info** **会计信息**
  - **Various Pointers** (for implementing scheduling queues)
  - **Pointer to enclosing process**: PCB, which process it belongs to





# Thread State

封装 并发性

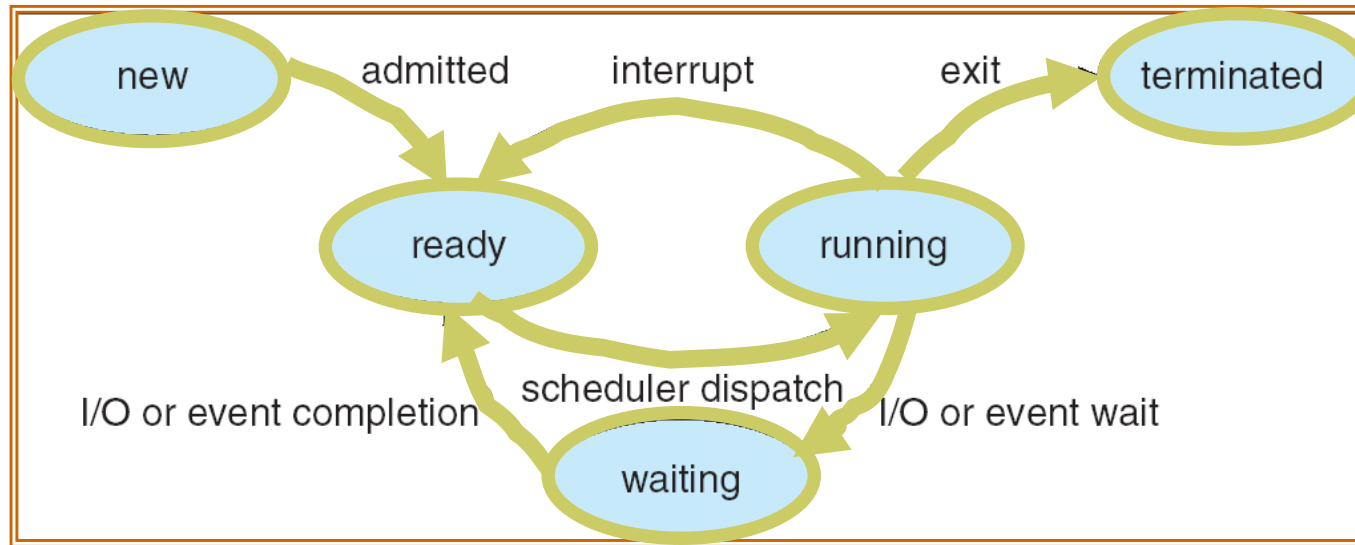
- Threads in a way encapsulate **concurrency**, which is the “**active**” component of a process
- Address space encapsulates **protection**: which can be considered as the “**passive**” part of a process
  - One program address space is different from that of another program so to keep buggy program from thrashing the entire system
- State **shared** by all threads in process/address space
  - Contents of memory (global variables, heap)
  - I/O state (file descriptors, network connections, etc.)
- State “**private**” to each thread
  - Kept in **TCB**  $\equiv$  **Thread Control Block**
  - CPU registers (including, program counter)
  - Execution stack (parameters, temporary variables, PC saved)







# Lifecycle of a Thread



- As a thread executes, it changes state:
  - **new**: The thread is being created
  - **ready**: The thread is waiting to run
  - **running**: Instructions are being executed
  - **waiting**: Thread waiting for some event to occur
  - **terminated**: The thread has finished execution
- “Active” threads are represented by their TCBs
  - TCBs organized into queues based on their states

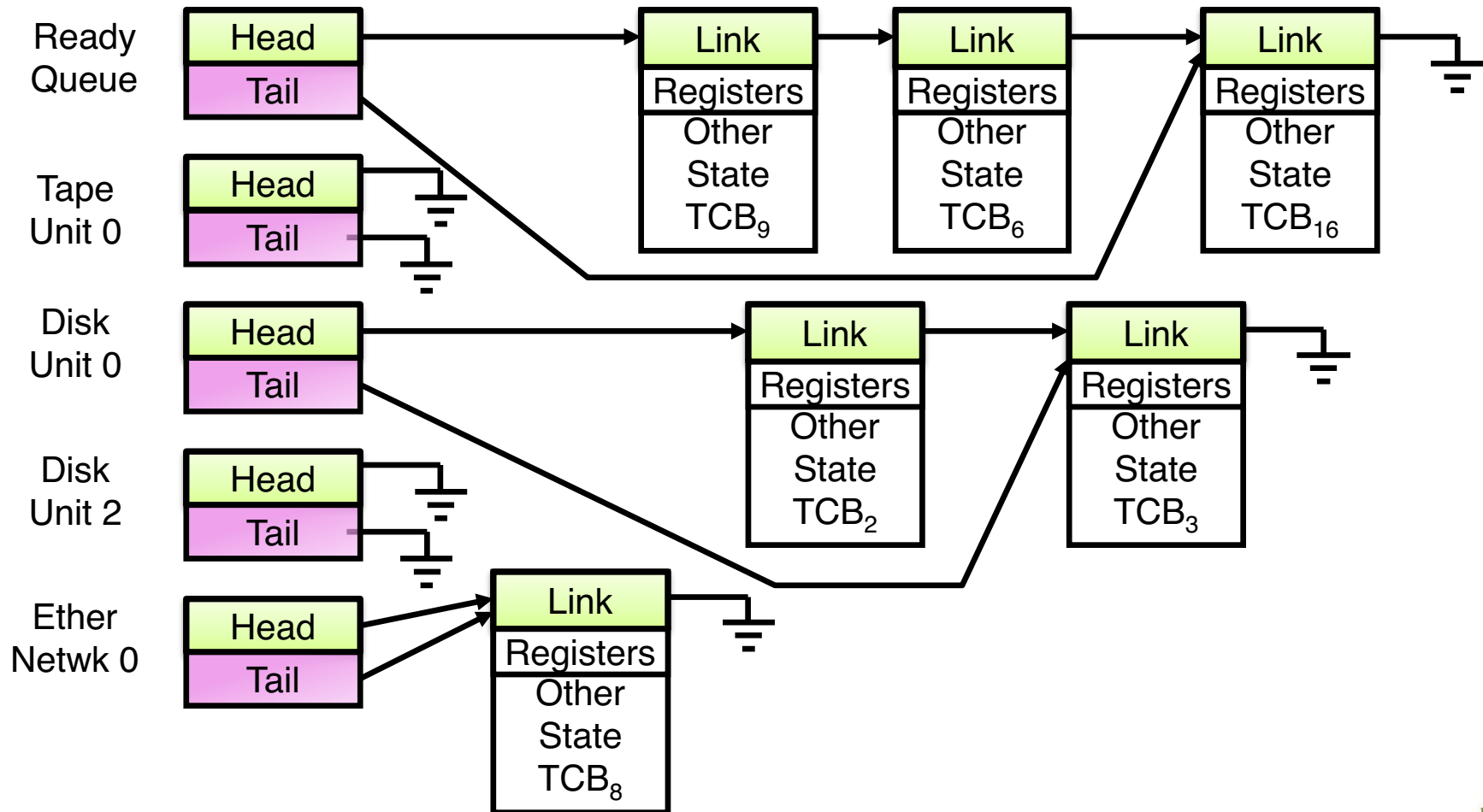




## Ready Queue And Various I/O Device Queues

Thread not running  $\Rightarrow$  TCB is in some other queue

- Queue for each device/signal/condition, each with its own scheduling policy

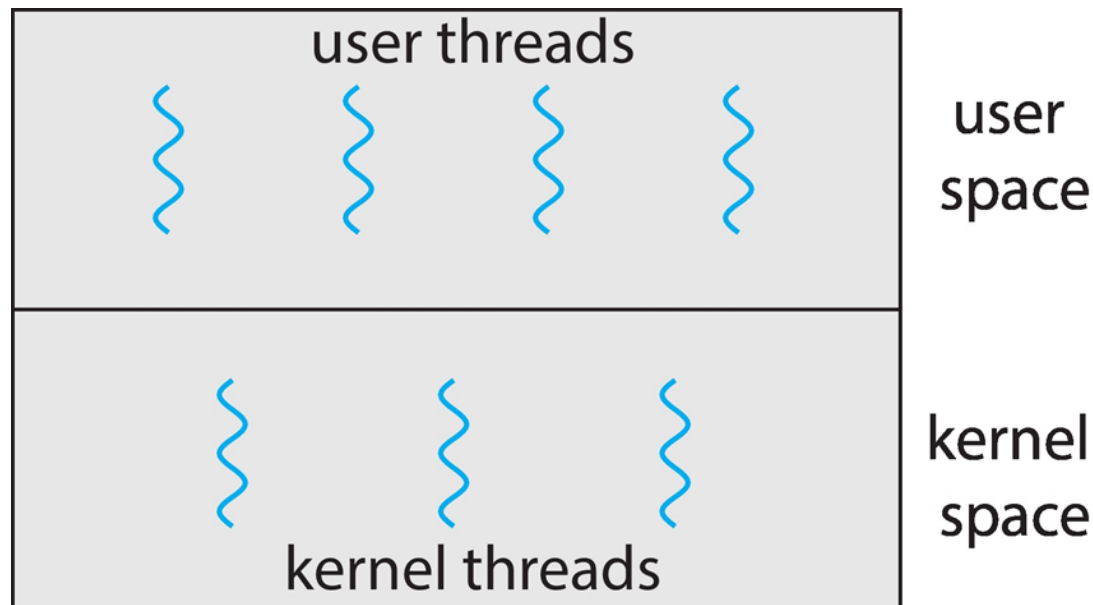




# User Threads and Kernel Threads

实体

- **User threads** – independently executable entity within a program, created and managed by **user-level threads library**
- **Kernel threads** – can be scheduled and execute on a CPU, supported and managed by the operating system
- Examples – virtually all general-purpose operating systems, including:
  - Windows, Linux, Mac OS X iOS, Android





# Multithreading Models 多线程模型

- **User-level threads** are visible only to programmers and unknown to the kernel. <sup>因此</sup> Consequently, they cannot be scheduled to run on a CPU. In another word, OS only manages and schedules **kernel threads**
- **Threads libraries** provide APIs for creating and managing user threads in programs. There are three primary thread libraries - **POSIX Pthreads**, **Windows threads** and **Java threads**, corresponding to the three most common APIs, e.g., Win32 API, POSIX API and Java API
- In a way, user-level threads provide the concurrent and modular entities in a program that the OS can take advantage of. Or if there is no user threads defined or specified in a program, there will be no kernel thread(s) that can be scheduled for the program execution <sup>模块实体</sup>

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





三种建立关系的方法

# Multithreading Models (Cont.)

映射

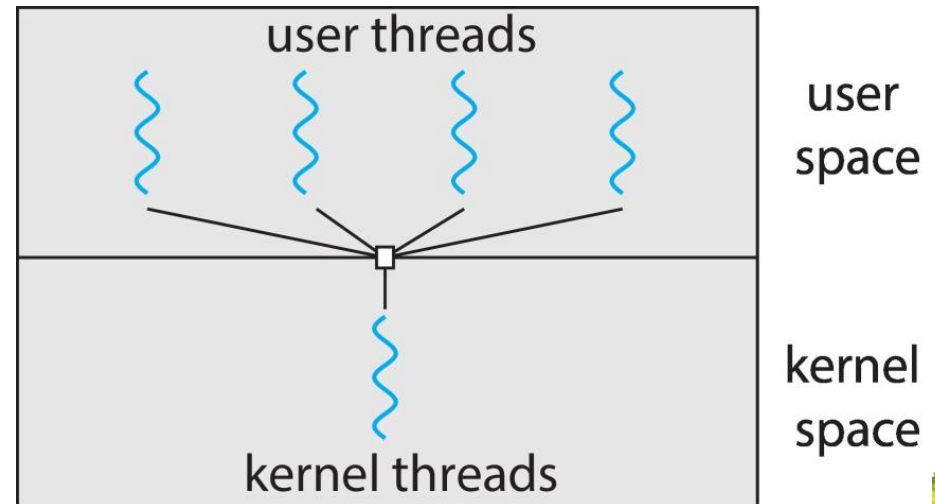
- Ultimately, a mapping must exist between user threads and kernel threads  
There are three common ways of establishing such a relationship:
  - **Many-to-One**: many user-level threads mapped to one kernel thread
  - **One-to-One**: each user-level thread mapped to a kernel thread – most common in modern operating system
  - **Many-to-Many**: many user-level threads mapped to many kernel (usually smaller number of) threads

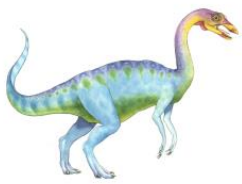




# Many-to-One

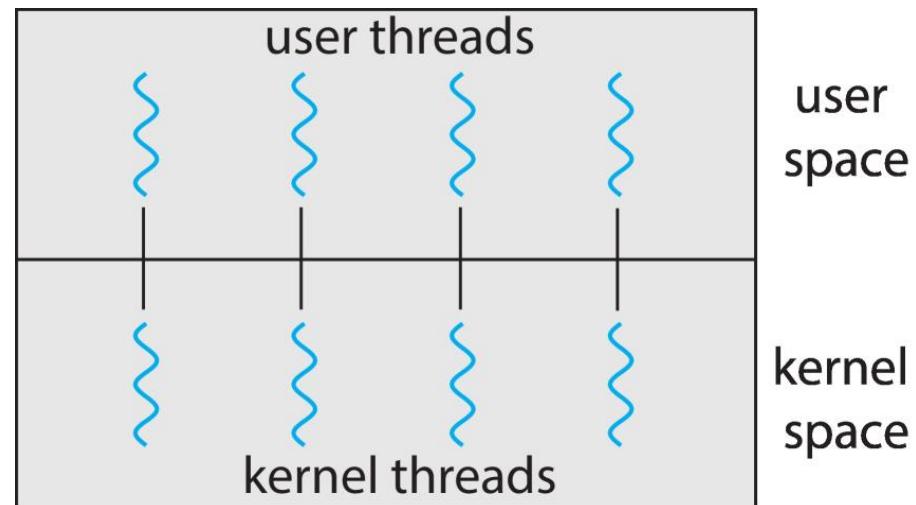
- Many user-level threads mapped to single kernel thread
- Which user thread is currently mapped to the kernel thread is a **scheduling problem**, known as **process-contention scope** or **PCS** (to be discussed in Chapter 5)  
进程争用范围
- One thread blocking (i.e., the kernel thread) causes all threads mapped to this thread to block, i.e., the entire process is blocked
- Multiple user threads can not run in parallel on multicore system because only one kernel thread can be active at a time
- Few systems currently use this model
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads





# One-to-One

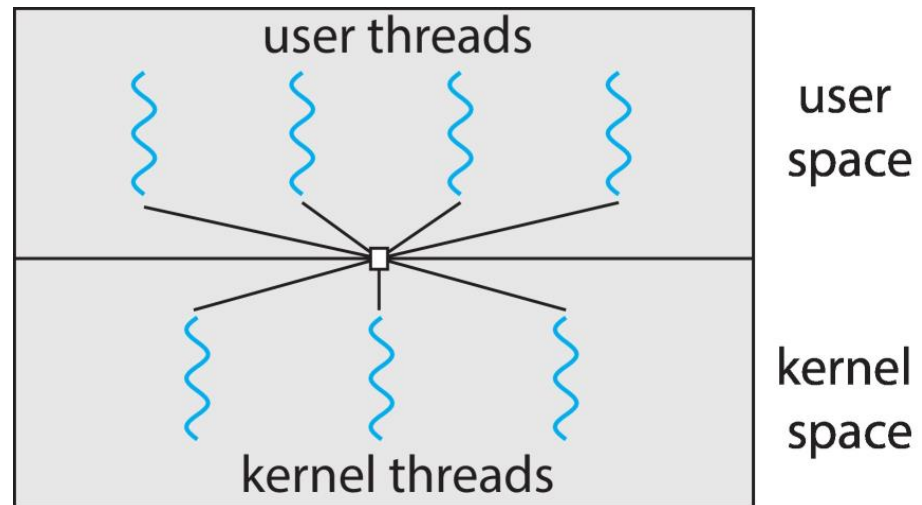
- Each user-level thread maps to one kernel thread
- It provides the maximum concurrency, and it also allows multiple threads to run in parallel on multiprocessor or multicore systems
- This implies that creating a user-level thread <sup>意味着</sup> mandates creating a <sup>要求</sup> corresponding kernel thread – the number of threads per process <sup>限制</sup> sometimes can be restricted due to overhead (kernel threads consume resources such as memory, I/O, etc.)
- Examples
  - Windows
  - Linux





# Many-to-Many Model

- ❑ Multiplexes many user-level threads to a smaller or equal number of kernel threads, specific to a particular application or machine
- ❑ This provides certain level of concurrency for a process execution – also **process-contention scope** or **PCS** is involved for scheduling
- ❑ This allows the operating system to create a <sup>足够的</sup>sufficient number of kernel threads in advance – **thread pool** 线程池
- ❑ Windows with the ThreadFiber package
- ❑ Otherwise not very common

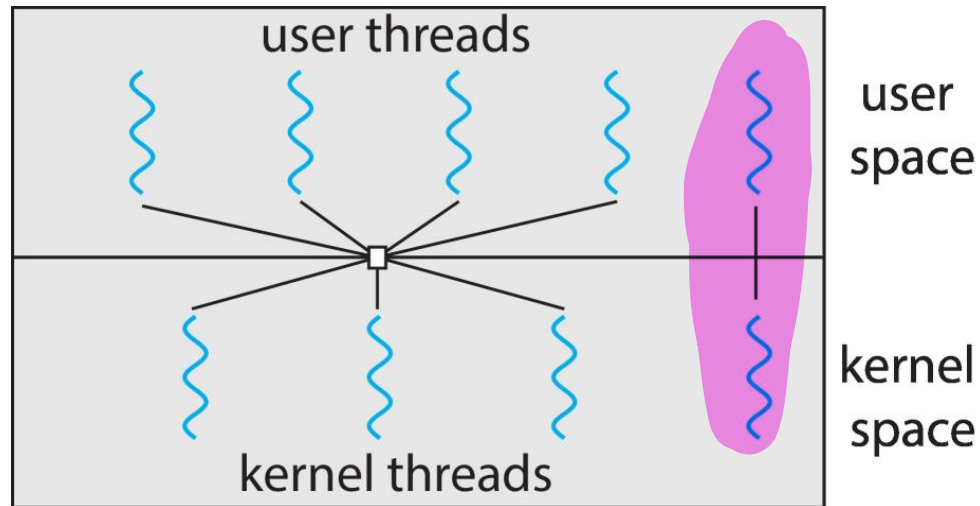






# Two-level Model

- Similar to the Many-to-Many model, except that it allows a user thread to be **bound** to a kernel thread





# Multithreading Mapping Models

- The many-to-many model is most flexible, but can be difficult to implement in practice 实现
  - For instance, an application may be allocated more kernel threads on a system with eight processing cores than on a system with four cores
- With an increasing number of processing cores in modern computer systems, limiting the number of kernel threads has become less important.
- Most operating systems now use the one-to-one model





# Threading Issues

语义

- Semantics of `fork()` and `exec()` system calls
- Signal handling
  - Synchronous and asynchronous
- Thread cancellation of target thread
  - Asynchronous or deferred cancellation

线程相关的问题包括以下几个方面：

## 1. `fork()` 和 `exec()` 系统调用的语义：

- `fork()`：这个系统调用用于创建一个新的进程。新进程是父进程的副本，拥有自己的线程和资源。对于线程来说，只有调用 `fork()` 的线程会被复制到新进程中，其他线程不会。
- `exec()`：这个系统调用用于替换当前进程的映像，加载一个新的程序。调用 `exec()` 后，当前进程的所有线程会被终止，新的程序会在该进程中运行。

## 2. 信号处理：

- 信号是操作系统用于通知进程发生特定事件的机制。线程可以接收和处理信号，但信号的处理可能会影响到整个进程中的所有线程。需要合理设计信号处理机制，以避免线程间的干扰。

## 3. 同步和异步：

- **同步**：在同步操作中，线程需要等待某个事件完成后才能继续执行。例如，线程在等待锁时会被阻塞，直到锁被释放。
- **异步**：在异步操作中，线程可以继续执行而不需要等待某个事件完成。例如，发起一个网络请求后，线程可以继续执行其他任务，等到请求完成时再处理结果。

## 4. 目标线程的取消：

- 线程取消是指终止一个正在运行的线程。可以通过两种方式进行：
  - **异步取消**：立即终止目标线程，可能导致资源未释放或数据不一致。
  - **延迟取消**：请求目标线程在下一个安全点（如完成某个操作）时自行终止，这样可以更安全地释放资源。





# Semantics of `fork()` and `exec()`

---

- ❑ Does **`fork()`** called by a thread duplicate only the calling thread or all threads of a process?
- ❑ Some UNIX systems have chosen to have two versions of **`fork()`** - If **`exec()`** is called immediately after forking, then duplicating all threads is unnecessary, as the program specified in **`exec()`** will replace the entire process. In this instance, duplicating only the calling thread is appropriate.
- ❑ **`exec()`** usually works as normal – replace the entire process including all threads





# Signal Handling

通知

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred. A signal may be received either **synchronously** or **asynchronously**, depending on the source and the reason
  - **Synchronous signals** include illegal memory access and division by 0
    - delivered to the same process that performed the operation that caused the signal (that is why they are considered **synchronous**)
  - When a signal is generated by an event external to a running process – **asynchronous signals**. Examples include terminating a process with specific keystrokes (such as **<control><C>**) and a timer expires
- Signals are handled in different ways
  - Some signals may be ignored, while others (for example, an illegal memory access) are handled by terminating the program

外部

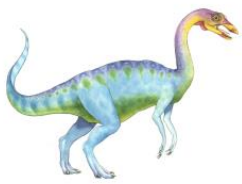




# Signal Handling (Cont.)

- A **signal handler** is used to process signals, following the pattern below
  - Signal is generated by a particular event (e.g., process termination)
  - Signal is delivered to a process
  - Signal is handled by one of two signal handlers, **default** or **user-defined**
- Every signal has **default handler** that kernel uses to handle that signal
  - **User-defined signal handler** can override default handler
- For a single-threaded process, signals are delivered to that process or thread
- Where should signal be delivered for a multi-threaded process?
  - 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





# Signal Handling (Cont.)

- The method for delivering a signal depends on the type of signal generated
  - For example, **synchronous signals** need to be delivered to the thread causing the signal and not to other threads in the process
  - Some **asynchronous signals** — such as a signal that terminates a process (**<control><C>**, for example)—should be sent to all threads
- UNIX function for delivering signal **kill(pid\_t pid, int signal)**, which specifies the process (**pid**) to which a particular signal (**signal**) is to be delivered
- POSIX Pthreads function allows a signal to be delivered to a specified thread (**tid**): **pthread\_kill(pthread\_t tid, int signal)**





# Thread Cancellation

- Terminating a thread before it has finished, and thread to be canceled is referred to as the **target thread**.
- The cancellation occurs in two different scenarios
  - **Asynchronous cancellation** <sup>异步 取消</sup> terminates the target thread **immediately**
  - **Deferred cancellation** <sup>延迟 取消</sup> The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion
- <sup>调用</sup> Invoking thread cancellation requests cancellation, **but actual cancellation depends on thread state and type**
- In Pthreads, if thread has cancellation disabled, cancellation remains pending until thread enables it – the default type is **deferred**

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







# Thread Cancellation (Cont.)

- Pthread code to create and cancel a thread:
  - Pthreads: POSIX standard for thread programming
  - Need to #include <pthread.h>

```
pthread_t tid;  
  
/* create the thread */  
pthread_create(&tid, 0, worker, NULL);  
  
. . .  
  
/* cancel the thread */  
pthread_cancel(tid);  
  
/* wait for the thread to terminate */  
pthread_join(tid, NULL);
```

Start routine  
↓





# Operating System Examples

---

- Windows Threads
- Linux Threads





# Windows Threads

- A Windows application runs as a separate process, and each process may contain one or more threads
- Windows API – primary API for Windows applications
- Windows uses the one-to-one mapping, where each user-level thread maps to an associated kernel thread
- The general components of a Window thread include:
  - A thread ID identifying the thread
  - Register set representing the state of the processor
  - A program counter (PC)
  - Separate user and kernel stacks when thread runs in user mode or kernel mode
  - Private data storage area used by run-time libraries and dynamic link libraries (DLLs)
- The register set, stacks, and private storage area are known as the context of the thread



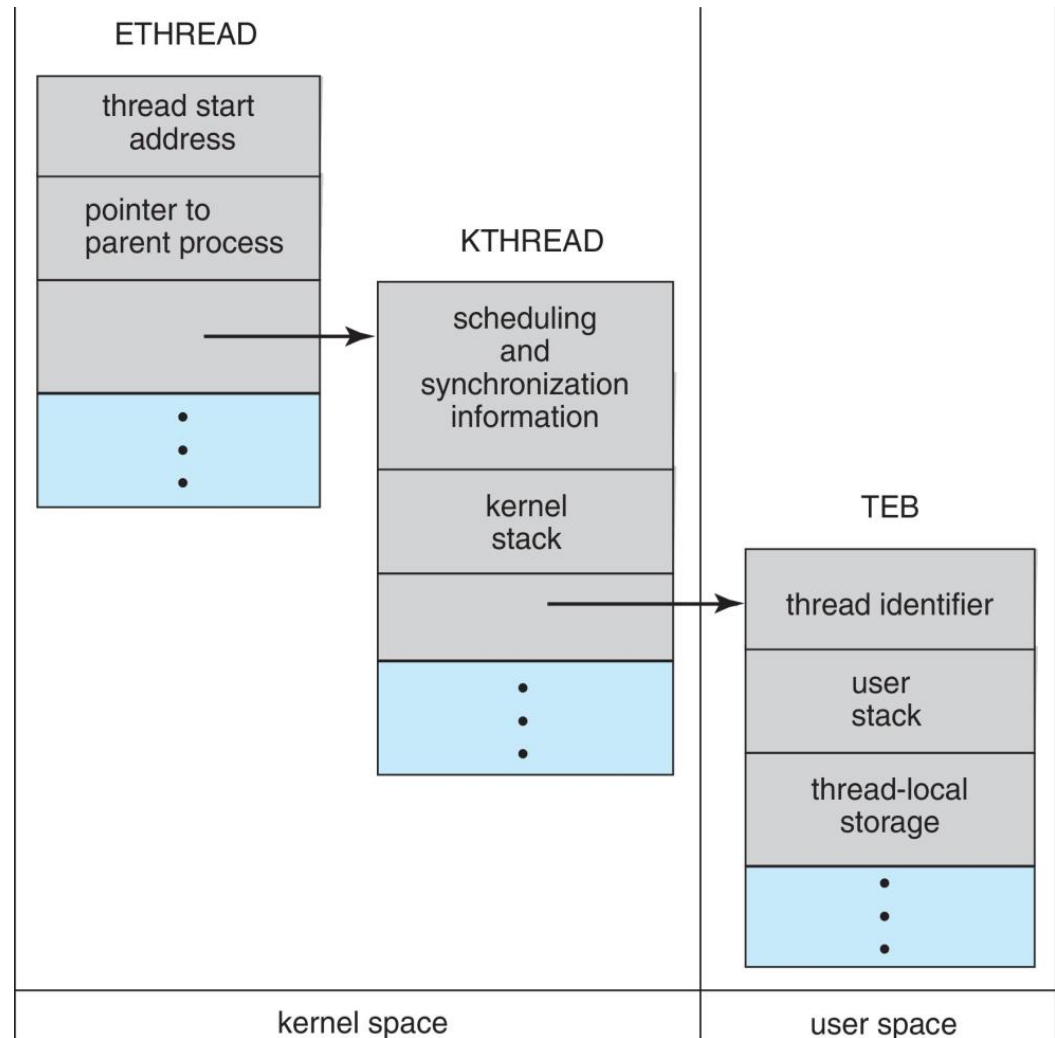


# Windows Threads (Cont.)

线程的主要数据:

The primary data structures of a thread include:

- ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, 执行
- 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





# Linux Threads

- ❑ Linux does not differentiate between threads and processes. To Linux, a thread is just a special kind of process, all referred as **tasks**
- ❑ A thread in Linux is a process that may share certain resources with other threads. Each thread has a unique **task\_struct** (illustrated in Chapter 3) and appears as a normal process - threads in Linux just happen to share resources, such as an address space, with other processes
- ❑ This approach to threads contrasts greatly with operating systems such as Microsoft Windows or Sun Solaris, which have explicit kernel support for threads (and sometimes call threads *lightweight processes*)
- ❑ For example, if a process consists of two threads
  - ❑ On Microsoft Windows, one PCB exists describing the shared resources such as address space or open files, and that, in turn, points to the two different threads. Each thread TCB describes the resources it alone possesses
  - ❑ In Linux, there are simply two **tasks** with normal **task\_struct** structures. The two processes are set up to share resources





# Linux Threads

- Threads are created in the same way as normal tasks, with the exception that the **clone()** system call is used to pass flags corresponding to the specific resources to be shared: 标志, 与...对应

```
clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);
```

- clone()** allows a child task to share the part of the execution context with a parent, such as address space, file descriptors, signal handler
- 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.   |

- Instead of copying all data structures like **fork()**, new task points to data structures of parent task, depending on set of flags passed to **clone()**
- A normal **fork()** can be implemented as **clone(SIGCHLD, 0);**

clone 更灵活, 不需要复制所有的进程



# End of Chapter 4

---

