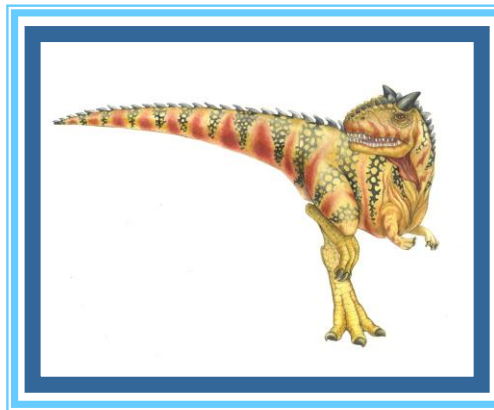
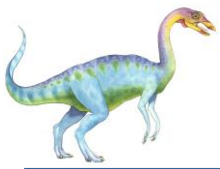


# Chapter 4: Threads

---





# Chapter 4: Threads

---

- Overview
- Multicore Programming
- Multithreading Models
- 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 examine issues related to multithreaded programming
- To cover operating system support for threads in Windows and Linux



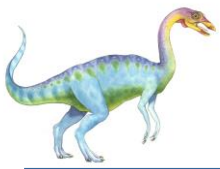


# Motivation

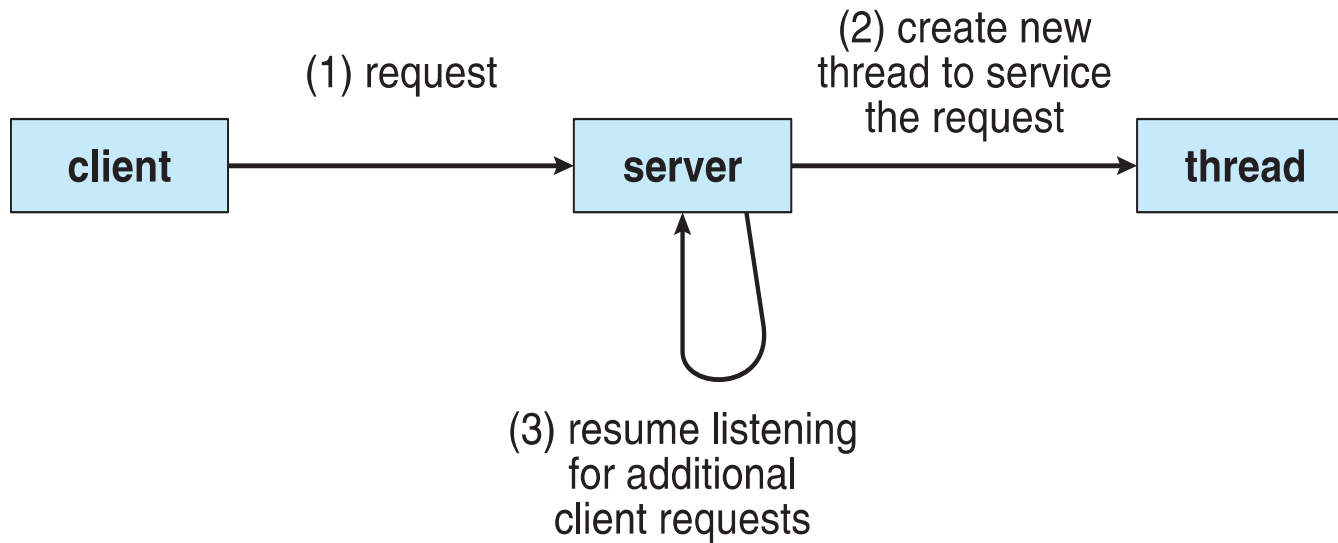
---

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded





# Multithreaded Server Architecture





# Benefits

---

- **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 passing
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching
- **Scalability** – process can take advantage of multiprocessor architectures



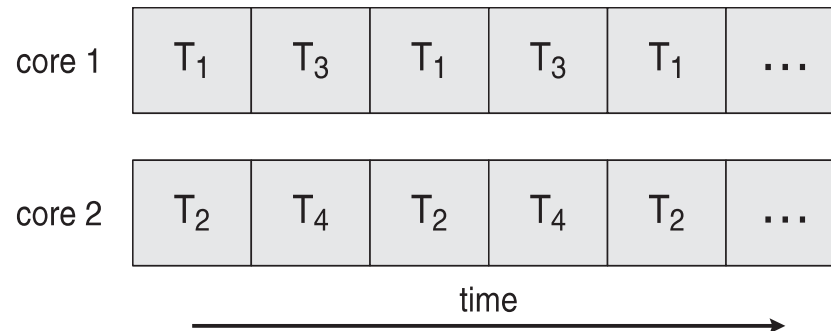


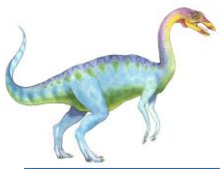
# Concurrency vs. Parallelism

## ■ Concurrent execution on single-core system:

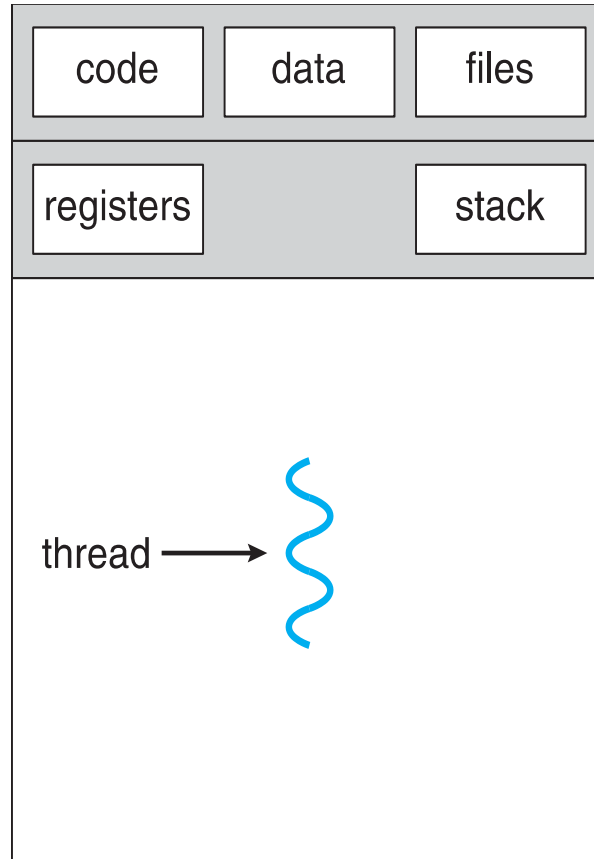


## ■ Parallelism on a multi-core system:

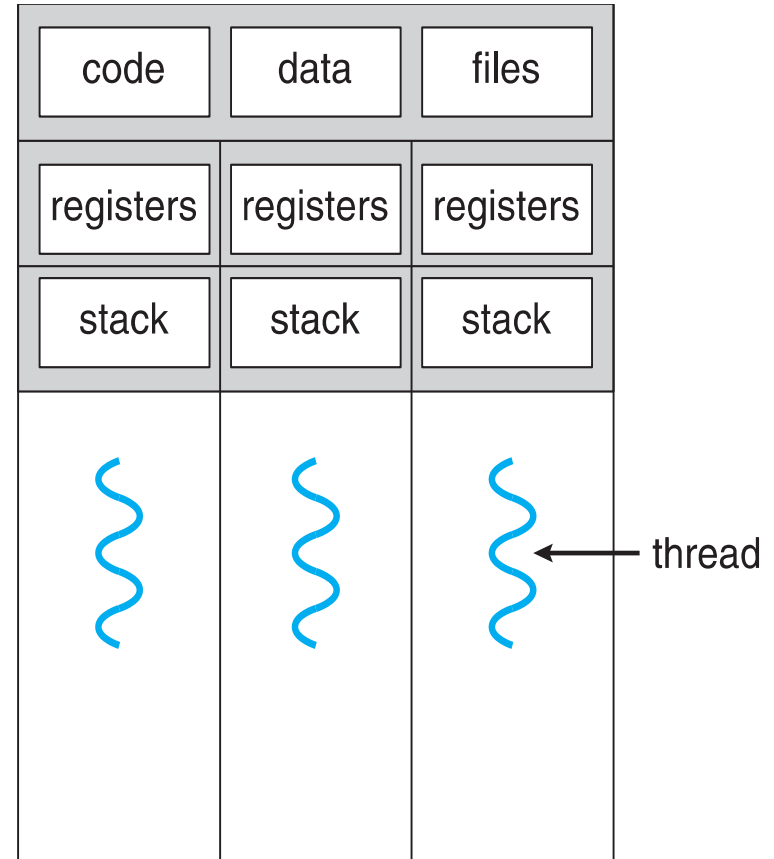




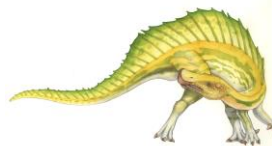
# Single and Multithreaded Processes



single-threaded process



multithreaded process







# Multithreading Models

---

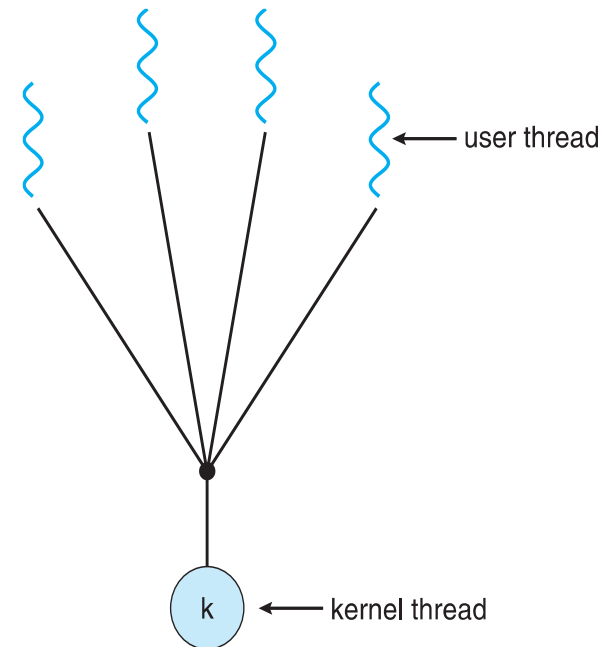
- Many-to-One
- One-to-One
- Many-to-Many





# Many-to-One

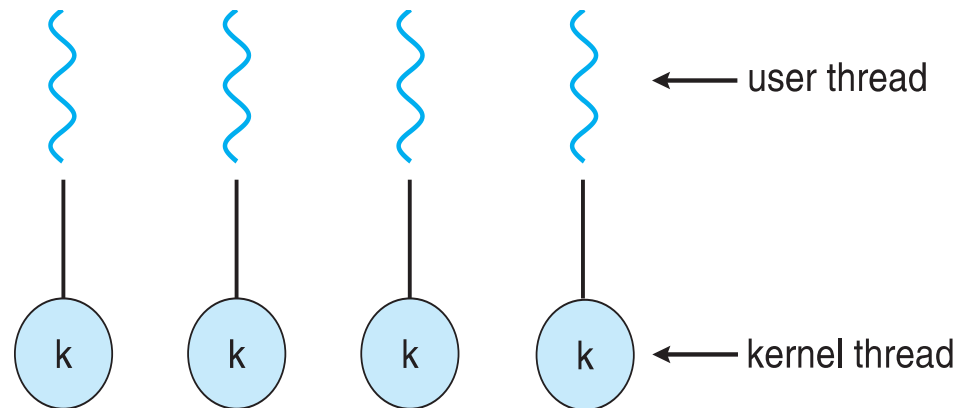
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable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





# One-to-One

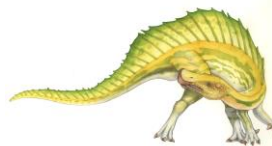
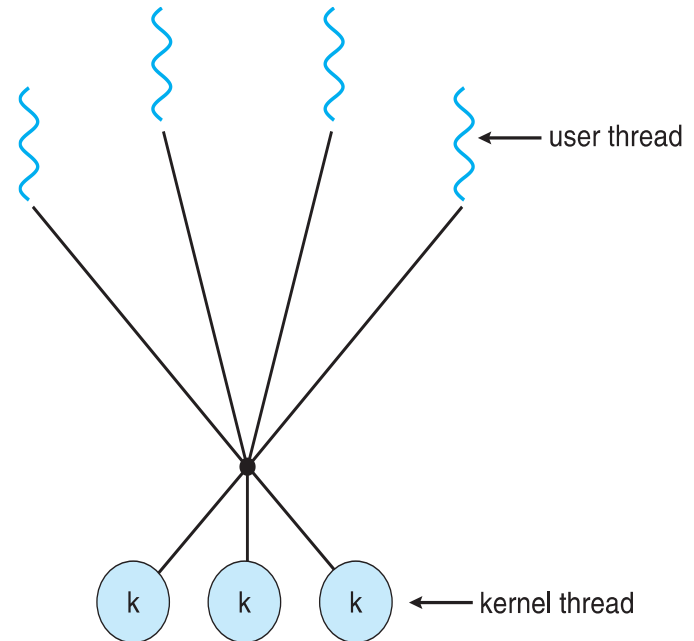
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later





# 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





# Threading Issues

---

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





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

---

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





# Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is **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);
```



# End of Chapter 4

---

