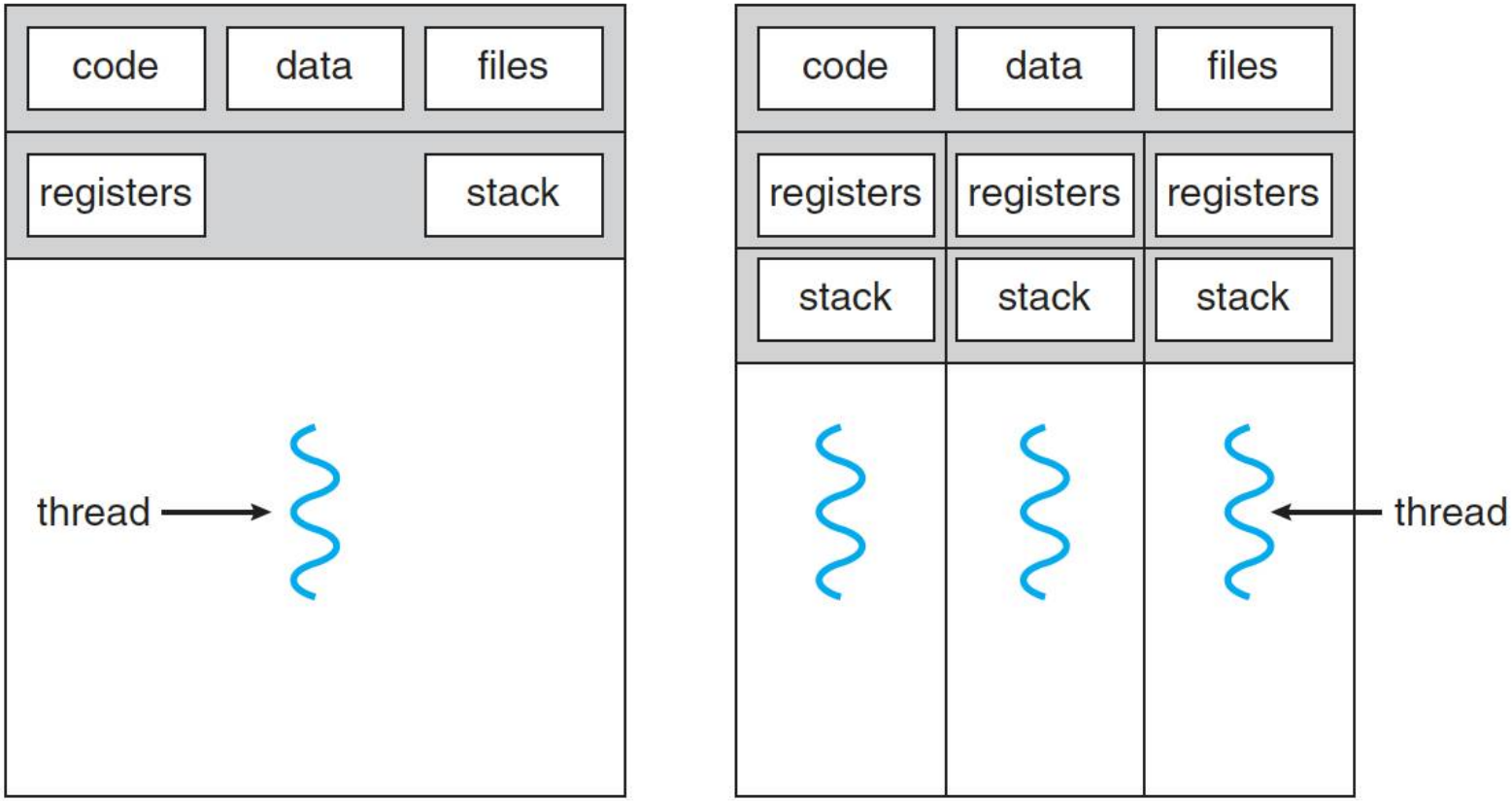
4 Threads

Thread

- A thread is a basic unit of CPU utilization
- It comprises a thread ID, a program counter, a register set and a stack
- A thread shares with other threads belonging to the same process its code section, data section, and other OS resources, such as open files and signals
- A traditional process has a single thread of control
- If a process has multiple threads of controls, it can perform more than one task at a time

Single-threaded vs. Multi-threaded Process



single-threaded process

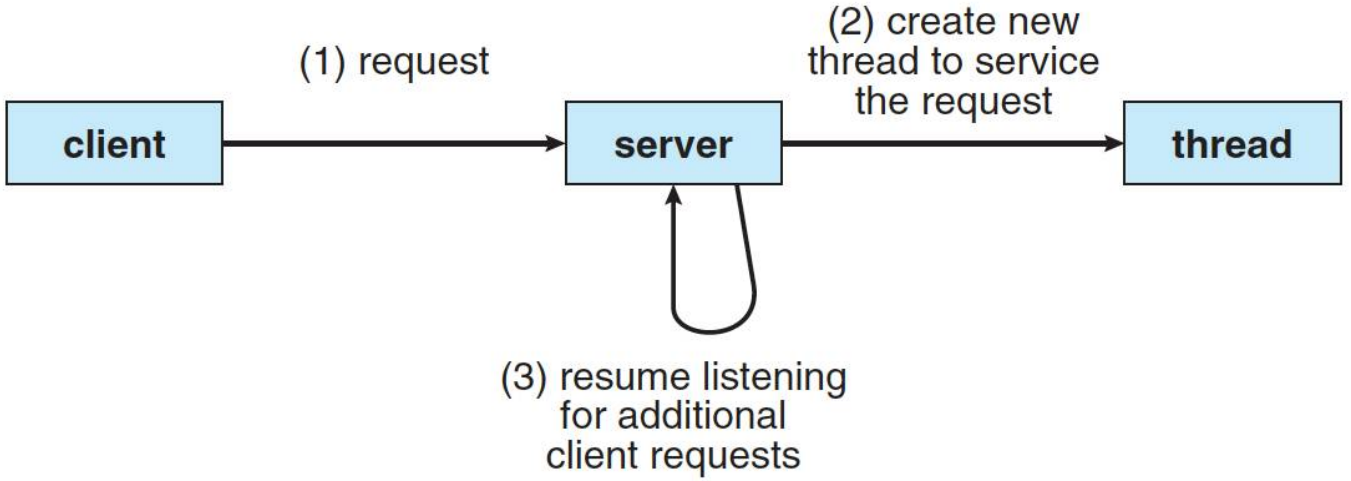
multithreaded process

Why Threads?

- A web browser might have one thread display images/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

- Multithreaded web server architecture

Multithreaded Server Architecture



Benefits

- Responsiveness
  - Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation
- Resource Sharing
  - threads share the memory and the resources of the process to which they belong by default
- Economy
  - it is more economical to create and context-switch threads
- Scalability
  - threads may be running in parallel on different processing cores

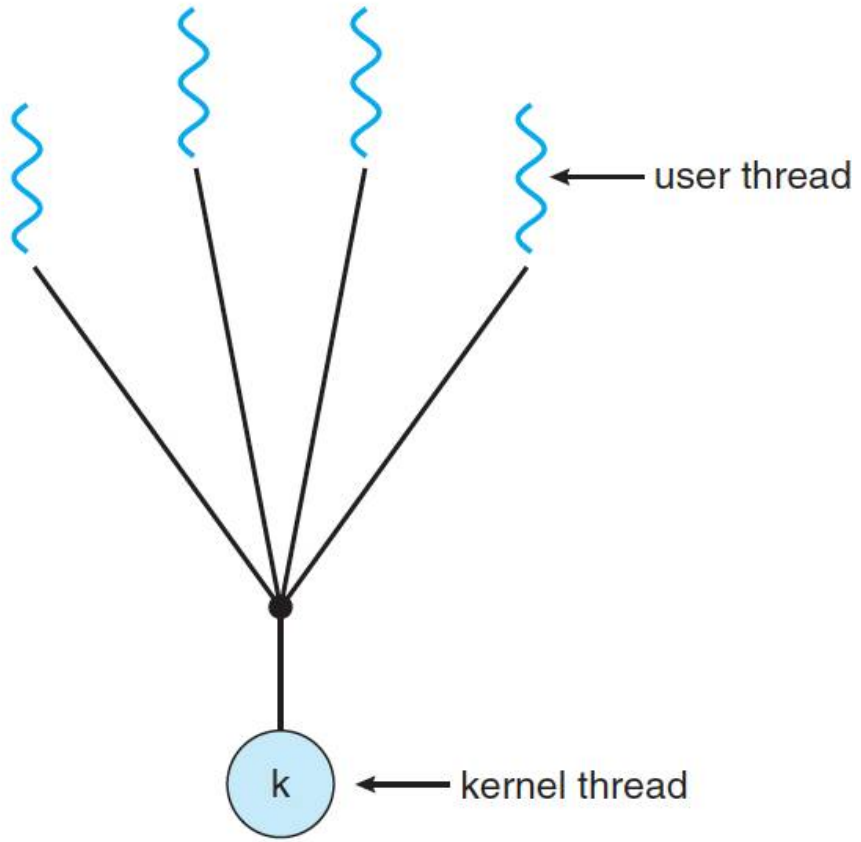
Multithreading Models

- Support for threads may be provided 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
- Kernel threads are supported and managed directly by the operating system
- Virtually all contemporary operating systems - including Windows, Linux, Mac OS X - support kernel threads
- A relationship must exist between user threads and kernel threads

Many-to-One

- Many user-level threads mapped to single kernel thread
  - Thread management is done by the thread library in user space, so it is efficient
  - However, the entire process will block if a thread makes a blocking system call.
  - Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

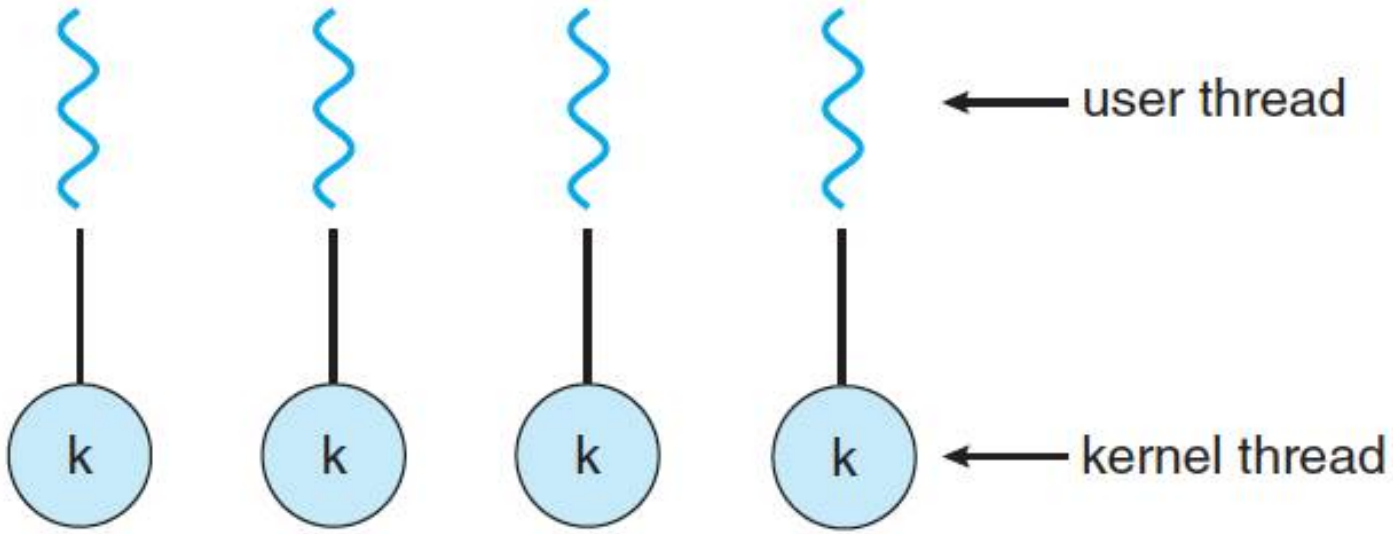
Many-to-One Model



One-to-One

- Each user-level thread maps to kernel thread
  - provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call
  - It also allows multiple threads to run in parallel on multiprocessors
  - 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
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

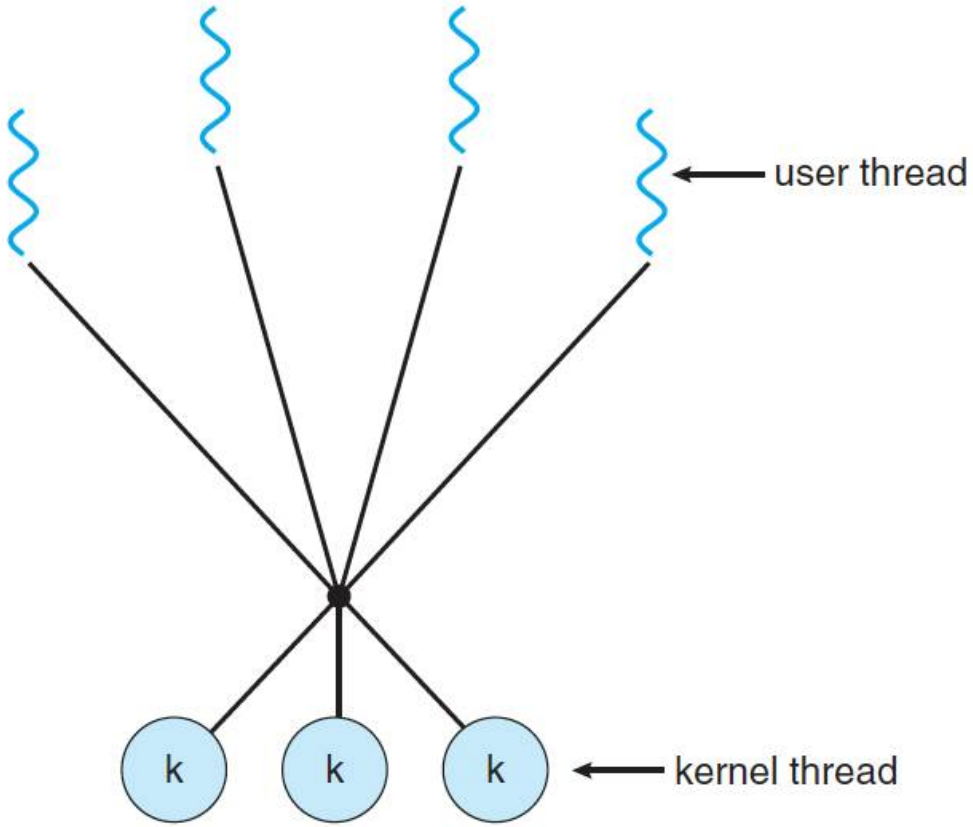
One-to-one Model



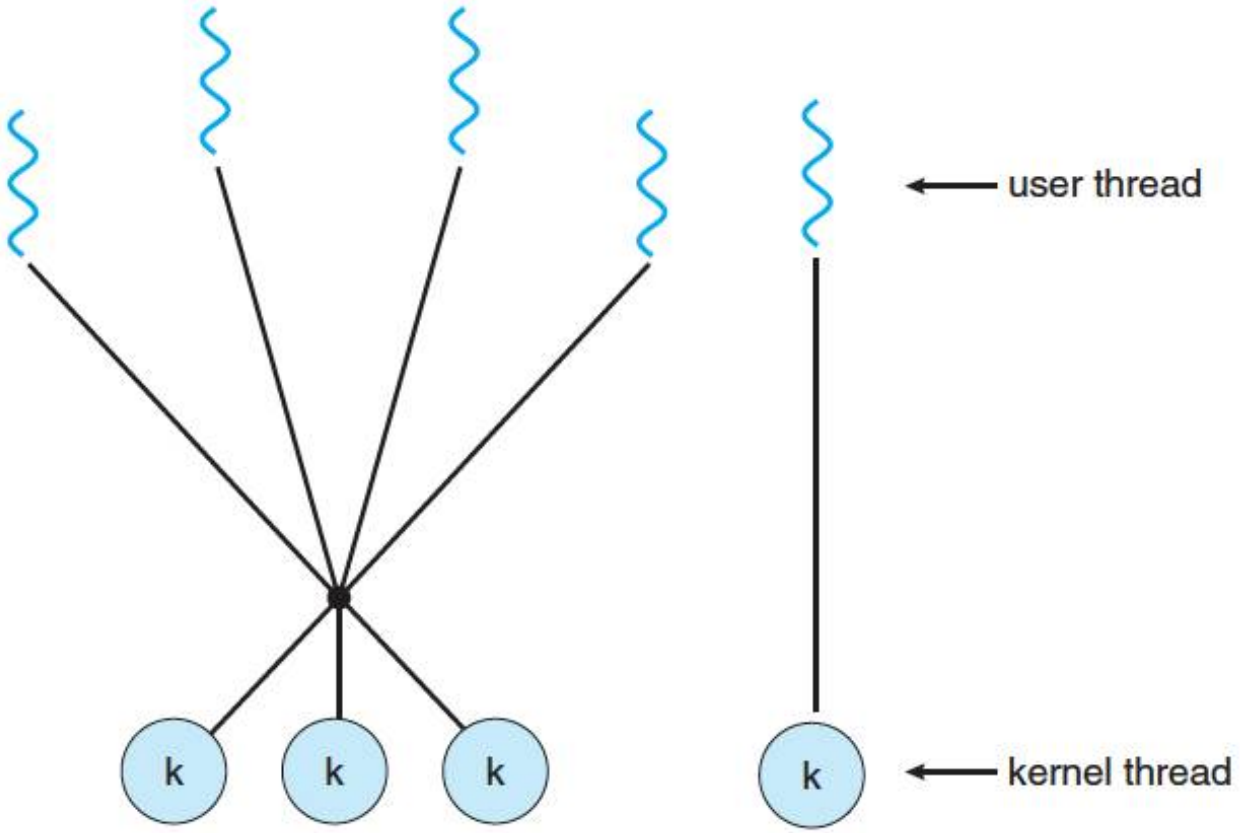
Many-to-Many Model

- The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads.
- Two-level model: Solaris prior to version 9
- Windows NT/2000 with the ThreadFiber package

Many-to-Many Model



Two-level Model



Thread Libraries

- A thread library provides the programmer with an API for creating and managing threads
- User-level library vs. kernel-level library
- Three main thread libraries are in use: POSIX Pthreads, Windows threads, Java threads

Pthreads

```
int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }

   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}
```

```
/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Windows Threads

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
      Sum += i;
   return 0;
}

int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;

   if (argc != 2) {
      fprintf(stderr,"An integer parameter is required\n");
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0 is required\n");
      return -1;
   }
```

```
/* create the thread */
ThreadHandle = CreateThread(
   NULL, /* default security attributes */
   0, /* default stack size */
   Summation, /* thread function */
   &Param, /* parameter to thread function */
   0, /* default creation flags */
   &ThreadId); /* returns the thread identifier */

if (ThreadHandle != NULL) {
    /* now wait for the thread to finish */
   WaitForSingleObject(ThreadHandle,INFINITE);

   /* close the thread handle */
   CloseHandle(ThreadHandle);

   printf("sum = %d\n",Sum);
}
}
```

Java Threads

```
class Sum
{
  private int sum;

  public int getSum() {
    return sum;
  }

  public void setSum(int sum) {
    this.sum = sum;
  }
}

class Summation implements Runnable
{
  private int upper;
  private Sum sumValue;

  public Summation(int upper, Sum sumValue) {
    this.upper = upper;
    this.sumValue = sumValue;
  }

  public void run() {
    int sum = 0;
    for (int i = 0; i <= upper; i++)
       sum += i;
    sumValue.setSum(sum);
  }
}
```

```
public class Driver
{
  public static void main(String[] args) {
    if (args.length > 0) {
      if (Integer.parseInt(args[0]) < 0)
        System.err.println(args[0] + " must be >= 0.");
      else {
        Sum sumObject = new Sum();
        int upper = Integer.parseInt(args[0]);
        Thread thrd = new Thread(new Summation(upper, sumObject));
        thrd.start();
        try {
           thrd.join();
           System.out.println
                   ("The sum of "+upper+" is "+sumObject.getSum());
        } catch (InterruptedException ie) { }
      }
    }
    else
      System.err.println("Usage: Summation <integer value>"); }
}
```

Threading Issues

- Semantics of fork() and exec() system calls
  - If one thread in a program calls fork(), does the new process duplicate all threads, or is the new process single-threaded
- Thread cancellation
  - The difficulty with cancellation occurs in situations where resources have been allocated to a canceled thread or where a thread is canceled while in the midst of updating data it is sharing with other threads
- Signal handling
  - a signal is typically delivered only to the first thread found that is not blocking it.
- Thread-local storage
- Scheduler activations