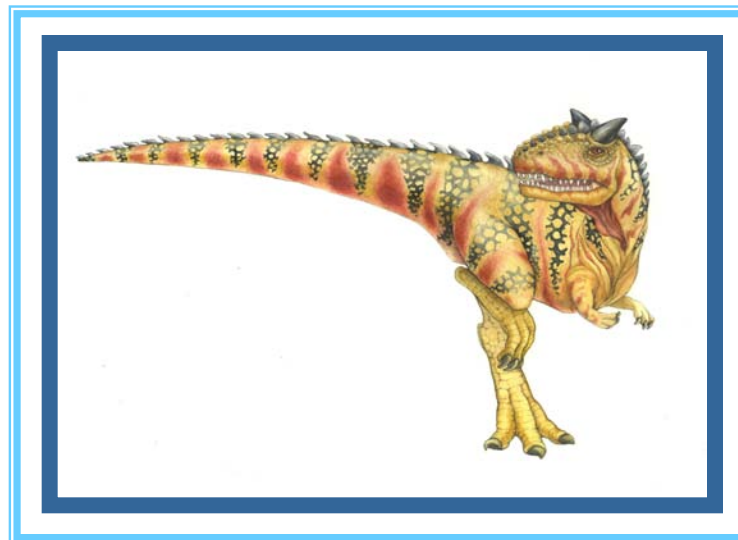


Chapter 4: Threads





Chapter 4: Threads

- Overview
- Multithreading Models
- Thread Libraries
- Threading Issues
- Operating System Examples
- Windows XP Threads
- Linux Threads





Objectives

- To introduce the notion of a thread — a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads, Win32, and Java thread libraries
- To examine issues related to multithreaded programming





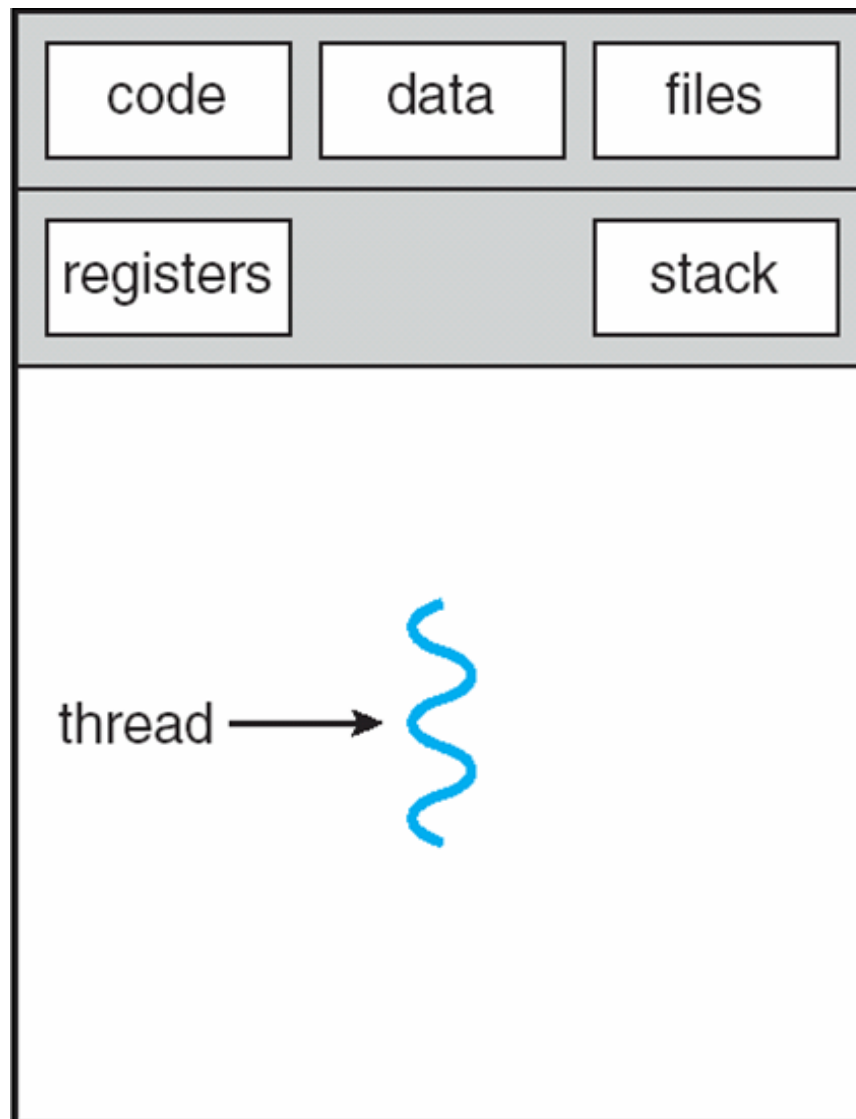
Motivation

- 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

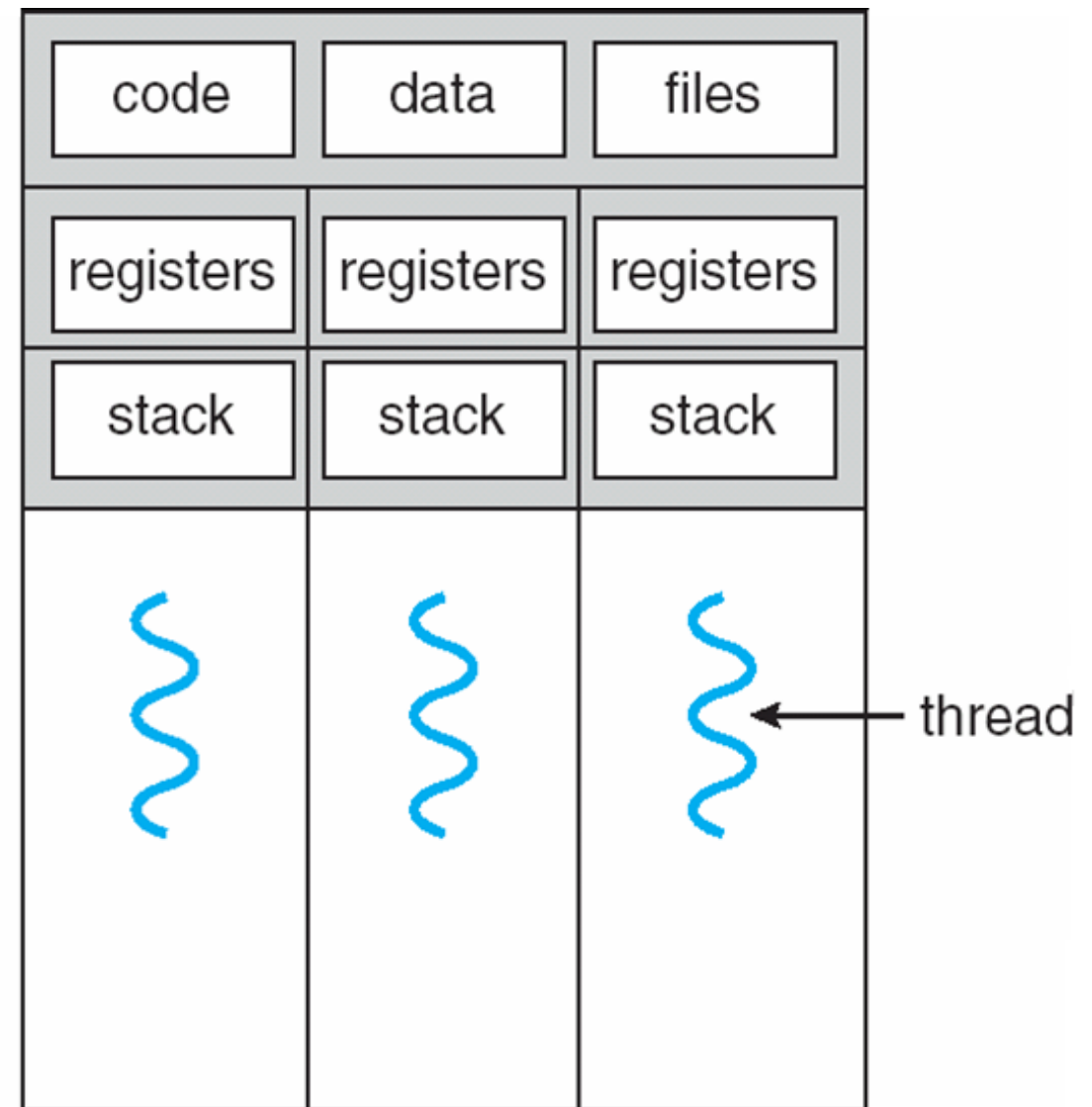




Single and Multithreaded Processes



single-threaded process



multithreaded process





Benefits

- **Responsiveness**
- **Resource Sharing**
- **Economy**
- **Scalability**





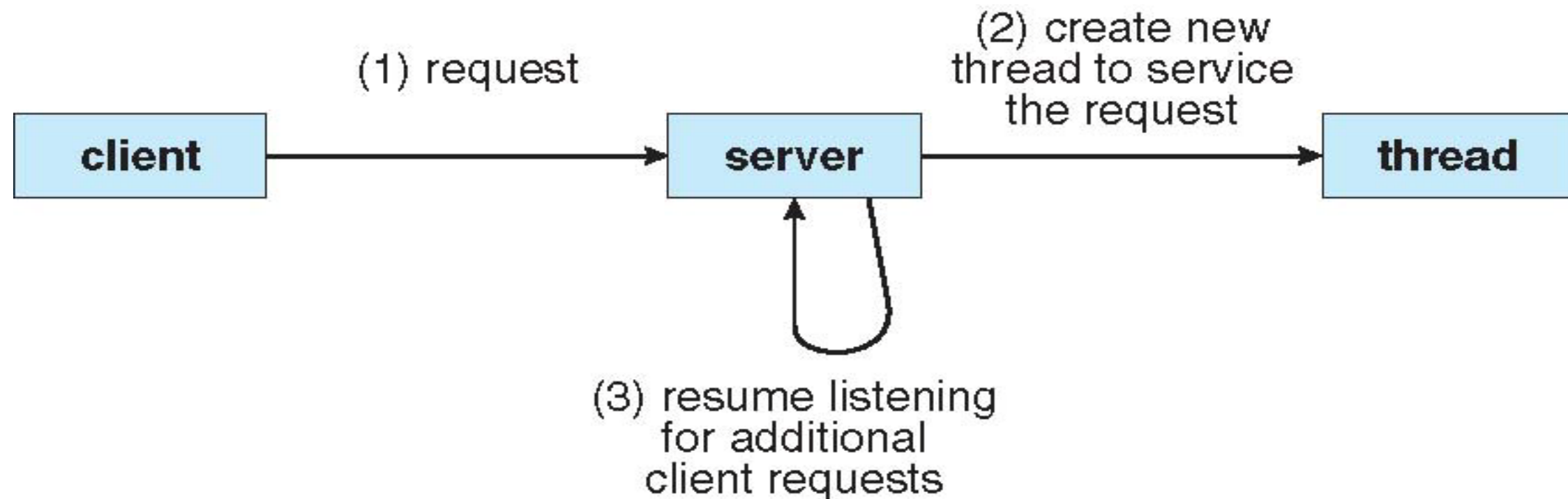
Multicore Programming

- Multicore systems putting pressure on programmers, challenges include:
 - **Dividing activities**
 - **Balance**
 - **Data splitting**
 - **Data dependency**
 - **Testing and debugging**



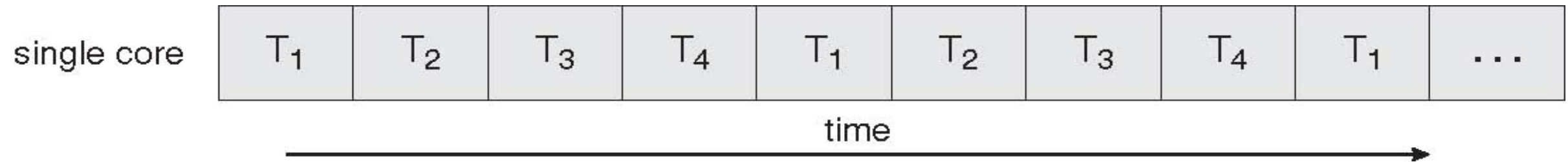


Multithreaded Server Architecture



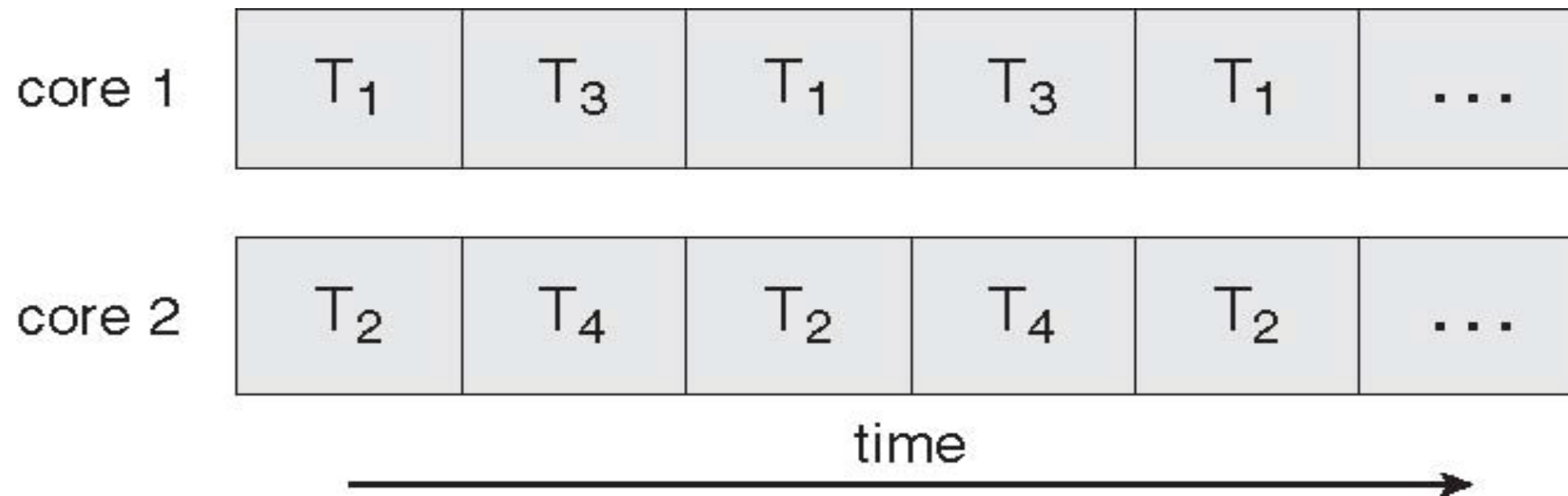


Concurrent Execution on a Single-core System





Parallel Execution on a Multicore System





User Threads

- Thread management done by user-level threads library
- Three primary thread libraries:
 - POSIX **Pthreads**
 - Win32 threads
 - Java threads





Kernel Threads

- Supported by the Kernel

- Examples
 - Windows XP/2000
 - Solaris
 - Linux
 - Tru64 UNIX
 - Mac OS X





Multithreading Models

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





Many-to-One

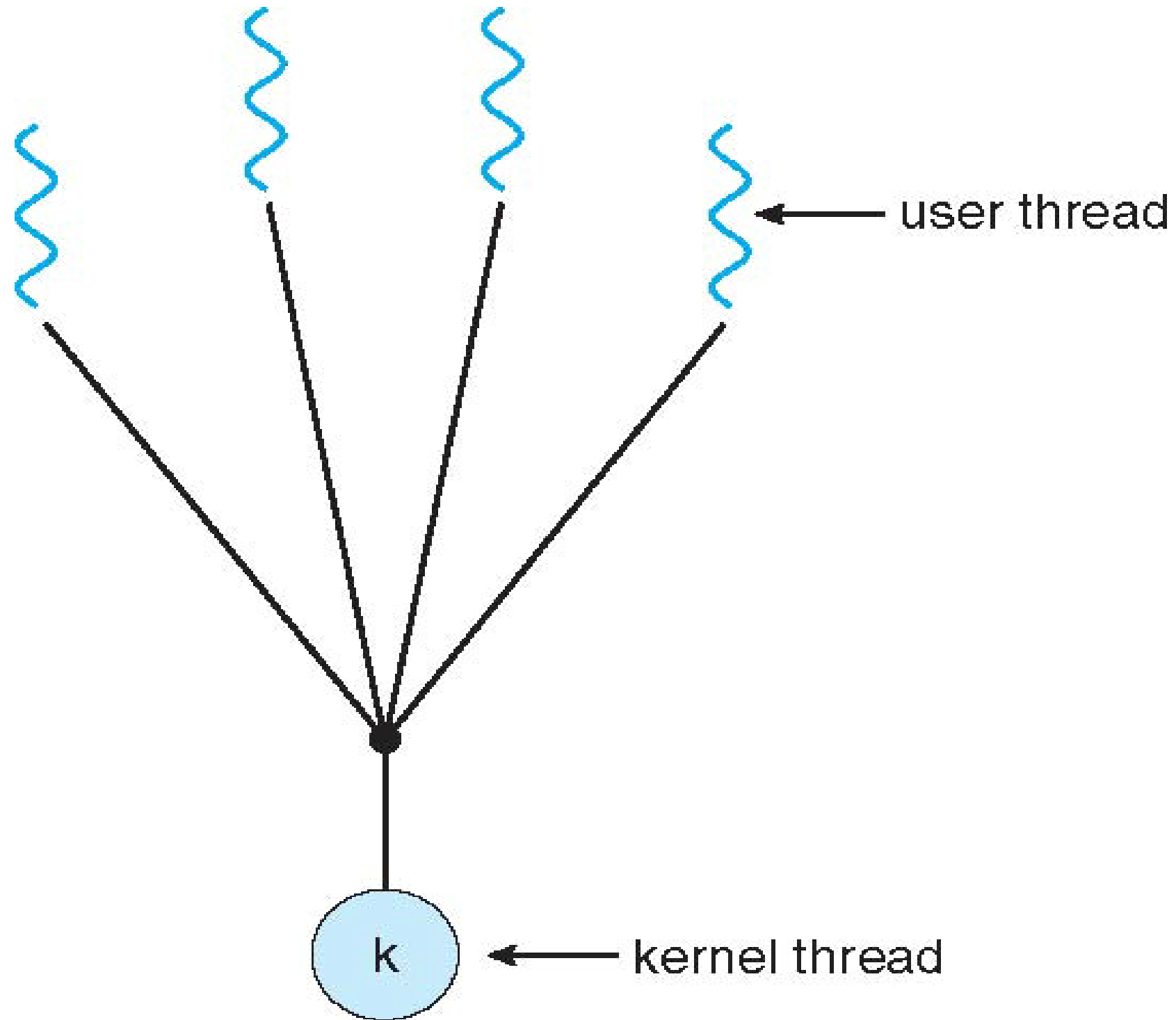
- Many user-level threads mapped to single kernel thread

- Examples:
 - **Solaris Green Threads**
 - **GNU Portable Threads**





Many-to-One Model





One-to-One

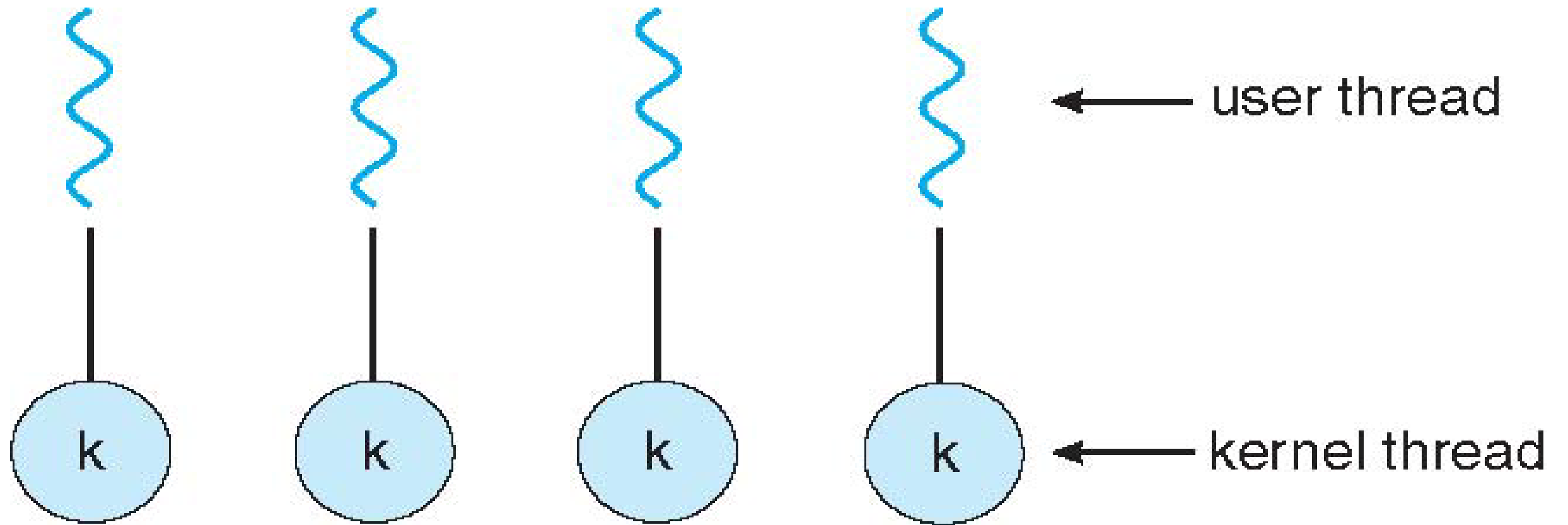
- Each user-level thread maps to kernel thread

- Examples
 - Windows NT/XP/2000
 - Linux
 - Solaris 9 and later





One-to-one Model





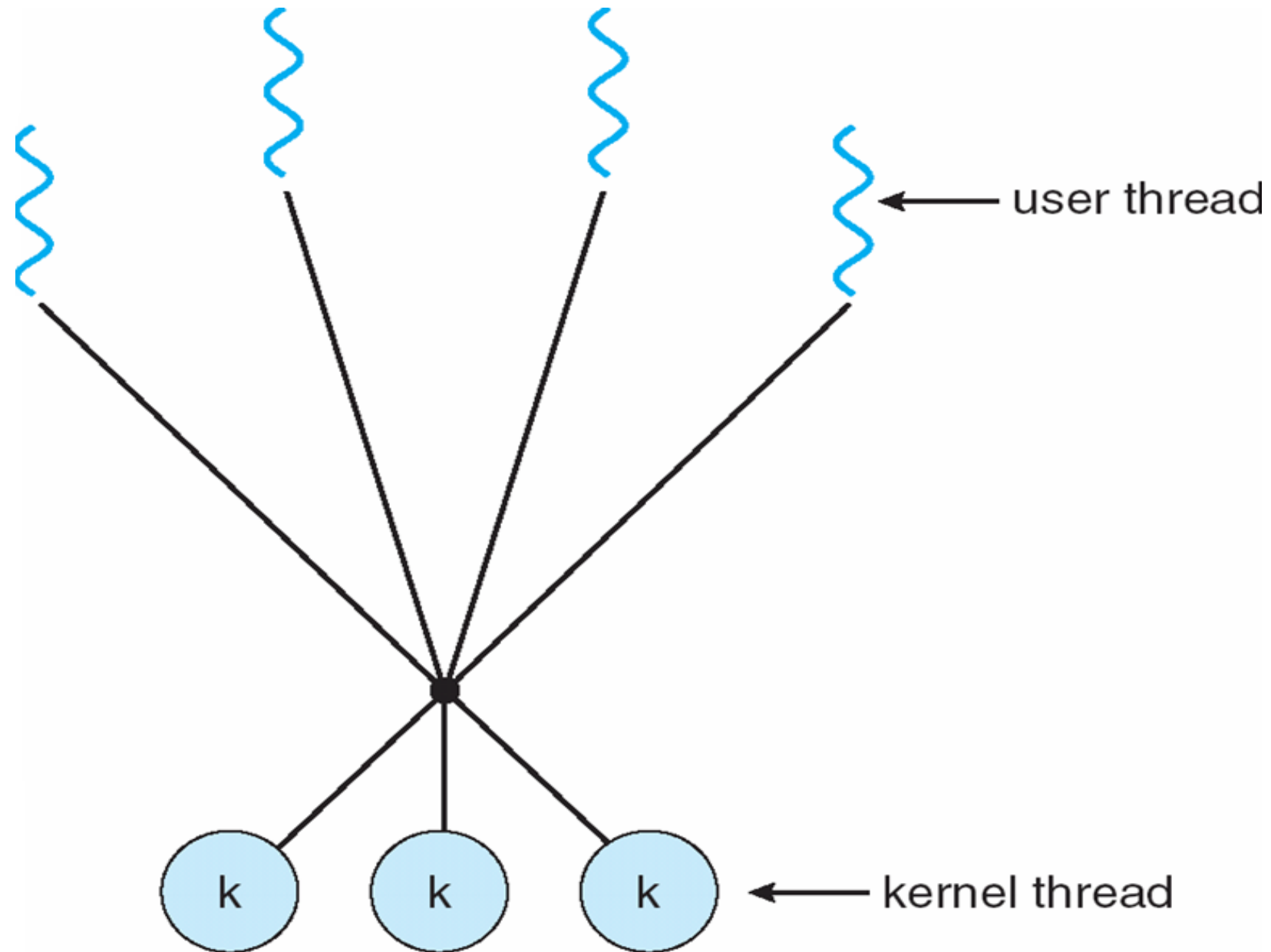
Many-to-Many Model

- 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 NT/2000 with the *ThreadFiber* package





Many-to-Many Model





Two-level Model

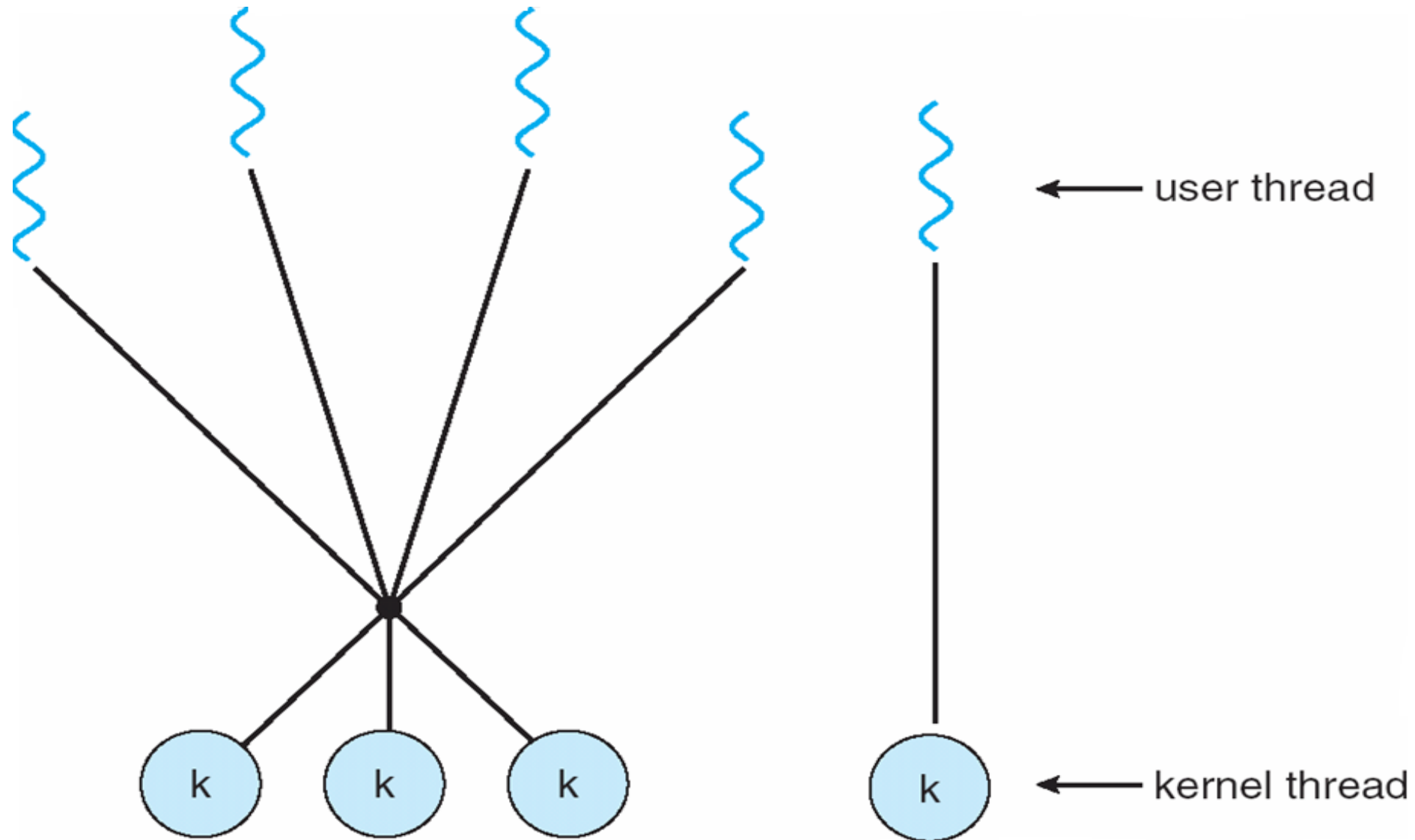
- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread

- Examples
 - IRIX
 - HP-UX
 - Tru64 UNIX
 - Solaris 8 and earlier





Two-level Model





Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads

- Two primary ways of implementing
 - Library entirely in user space
 - Kernel-level library supported by the OS





Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)





Pthreads Example

```
#include <pthread.h>
#include <stdio.h>

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

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;
    }
}
```





Pthreads Example (Cont.)

```
/* get the default attributes */
pthread_attr_t attr;
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);
}
```

Figure 4.9 Multithreaded C program using the Pthreads API.





Win32 API Multithreaded C Program

```
#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;
    /* perform some basic error checking */
    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;
    }
}
```





Win32 API Multithreaded C Program (Cont.)

```
// 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);
}
}
```

Figure 4.10 Multithreaded C program using the Win32 API.





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:
 - Extending Thread class
 - Implementing the Runnable interface





Java Multithreaded Program

```
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);
    }
}
```





Java Multithreaded Program (Cont.)

```
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 {
                // create the object to be shared
                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>");
    }
}
```

Figure 4.11 Java program for the summation of a non-negative integer.





Threading Issues

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





Threading Issues (Cont.)

- **Thread pools**
- **Thread-specific data**
 - Create Facility needed for data private to thread
- **Scheduler activations**





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

- Does `fork()` duplicate only the calling thread or all threads?





Thread Cancellation

- Terminating a thread before it has finished

- Two general approaches:
 - **Asynchronous cancellation** terminates the target thread immediately.
 - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled.





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
- Options:
 - 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 Pools

- Create a number of threads in a pool where they await work

- Advantages:
 - Usually slightly faster to service a request with an existing thread than create a new thread
 - Allows the number of threads in the application(s) to be bound to the size of the pool





Thread Specific Data

- 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)





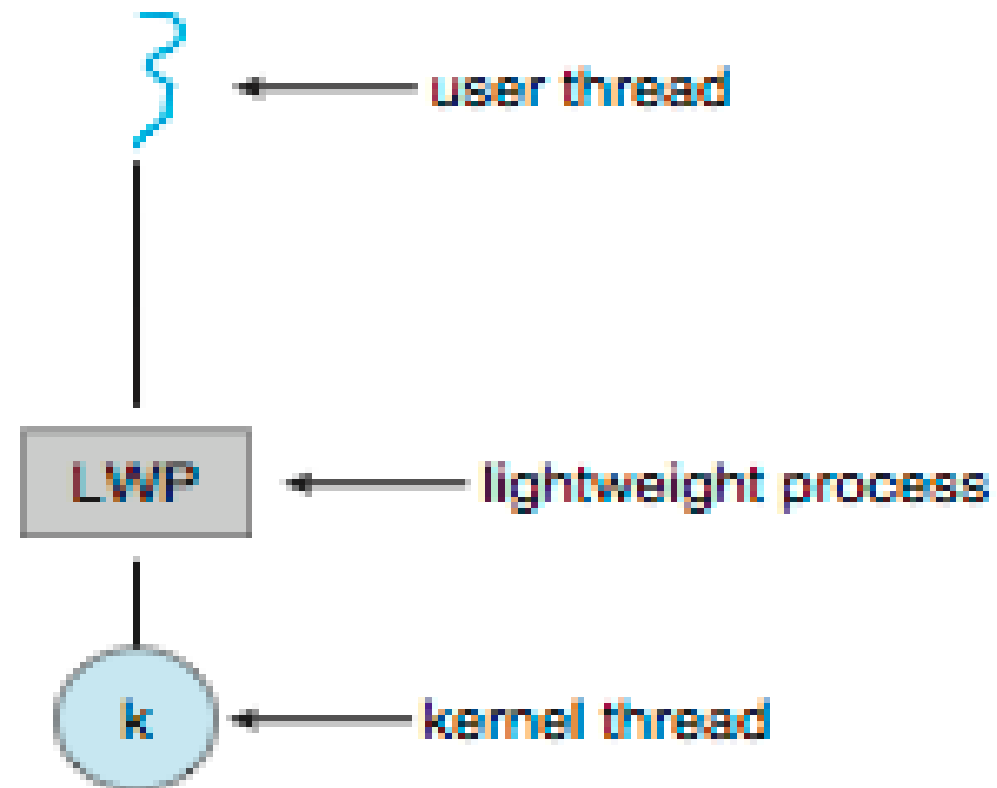
Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the thread library
- This communication allows an application to maintain the correct number kernel threads





Lightweight Processes





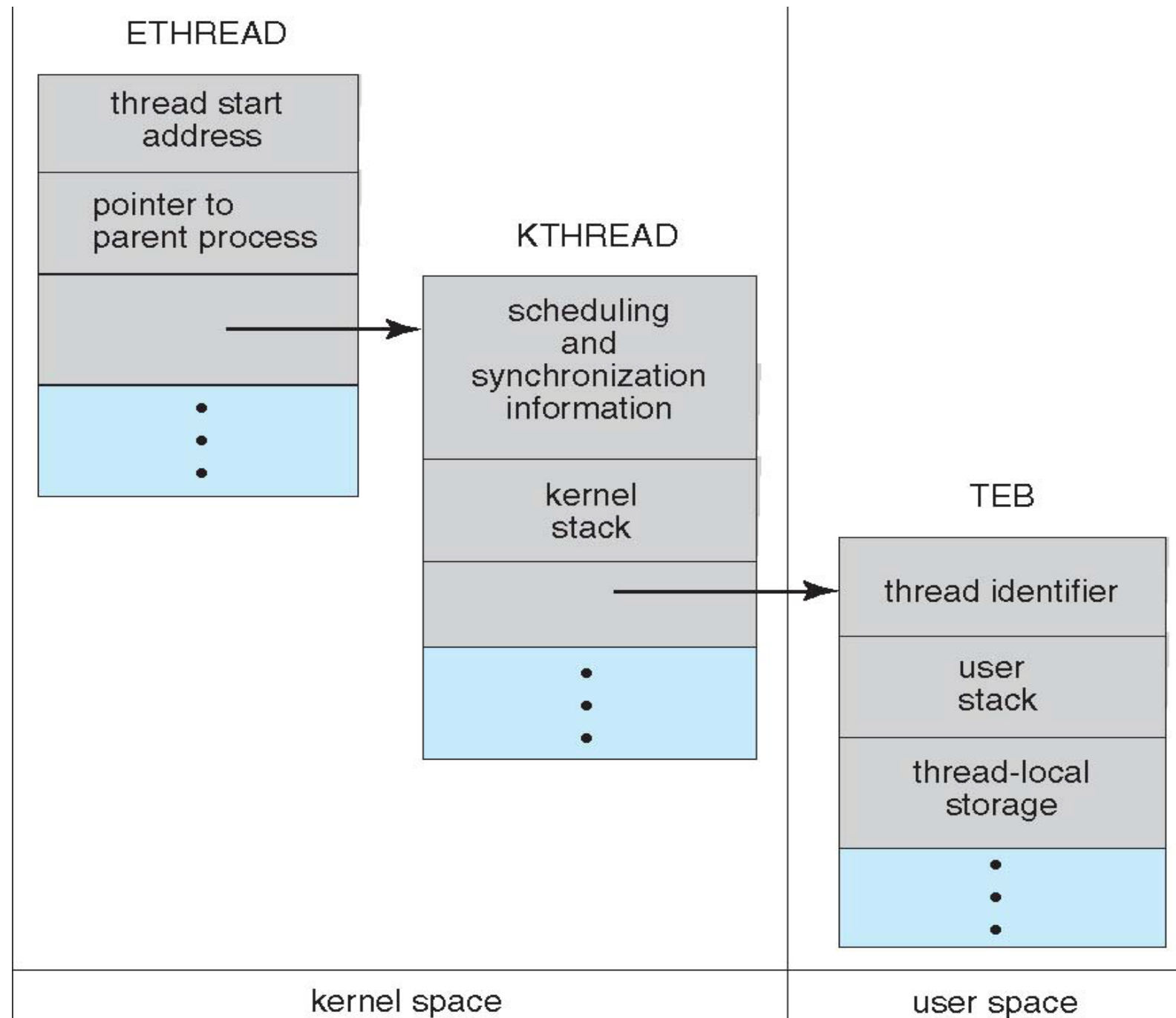
Operating System Examples

- Windows XP Threads
- Linux Thread





Windows XP Threads Data Structures





Windows XP Threads

- Implements the one-to-one mapping, kernel-level
- Each thread contains
 - A thread id
 - Register set
 - Separate user and kernel stacks
 - Private data storage area
- The register set, stacks, and private storage area are known as the **context** of the threads
- The primary data structures of a thread include:
 - ETHREAD (executive thread block)
 - KTHREAD (kernel thread block)
 - TEB (thread environment block)





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





Linux Threads

- `fork()` and `clone()` system calls
- Doesn't distinguish between process and thread
 - Uses term *task* rather than thread
- `clone()` takes options to determine sharing on process create
- `struct task_struct` points to process data structures (shared or unique)

flag	meaning
<code>CLONE_FS</code>	File-system information is shared.
<code>CLONE_VM</code>	The same memory space is shared.
<code>CLONE_SIGHAND</code>	Signal handlers are shared.
<code>CLONE_FILES</code>	The set of open files is shared.



End of Chapter 4

