Chapter 4: Threads

Chapter 4: Threads

- Overview
- Multithreading Models
- Threading Issues
- Pthreads
- Windows XP Threads
- Linux Threads
- Java Threads

Suppose you are developing a Web browser

```
main(){
        while(1){
                RetrieveData();    // Block for 1 second
                DisplayData();     // Block for 1 second
                GetInputEvents();  // Block for 1 second
        }
}
```

Now what if you want the program to be more responsive?

```
main(){
        while(1){
                RetrieveALittleData();    // Block for 0.1 second
                DisplayALittleData();     // Block for 0.1 second
                GetAFewInputEvents();     // Block for 0.1 second
        }
}
```

What if you want it to be even more responsive?

```
main(){
   while(1){
        if(CheckData()==True){
                RetrieveALittleData();  // 0.1 second
                DisplayALittleData();   // 0.1 second
        }
        if(CheckInputEvents()==True){
                GetAFewInputEvents();   // 0.1 second
        }
   }
}
```

Problem: A lot checks, not efficient. But still not responsive!

To make it responsive enough, we need to

- Break the operations into very very small pieces;

These two requirements conflict with each other!

- However, to be efficient enough, we want to execute code in large pieces.
- More precisely, we want to SCHEDULE these operations in our own program code.
- Leave the tedious work to the OS which schedules them in *Threads*!

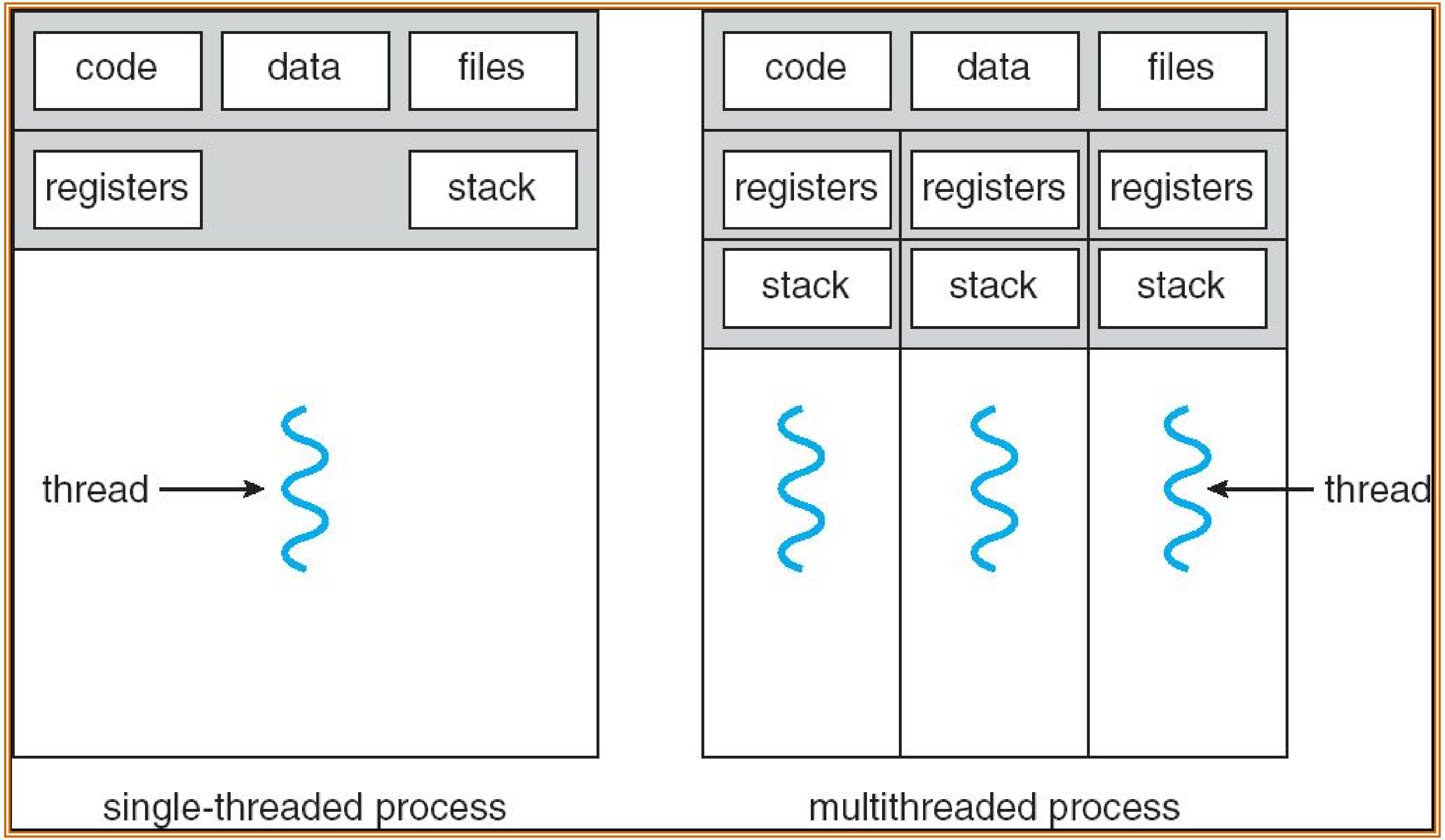
Multi-thread version of the program

```
main(){
      CreateThread (RetrieveData());
      CreateThread (DisplayData());
      CreateThread (GetInputEvents());
      WaitForThreads();
}
// Each thread routine enters a loop.
void RetrieveData(){
   while(1){
     retrieveData();
     …
   }
}
….
```

```
void DisplayData(){
  while(1){
    displayData();
    …
  }
}
```

……

Single and Multithreaded Processes



Benefits

- **Responsiveness**
  interactive applications
- **Resource Sharing**
  memory for code and data can be shared.
- **Economy**
  creating processes are more expensive.
- **Utilization of MP Architectures**

  multi-threading increases concurrency.

Concurrency vs. Parallelism

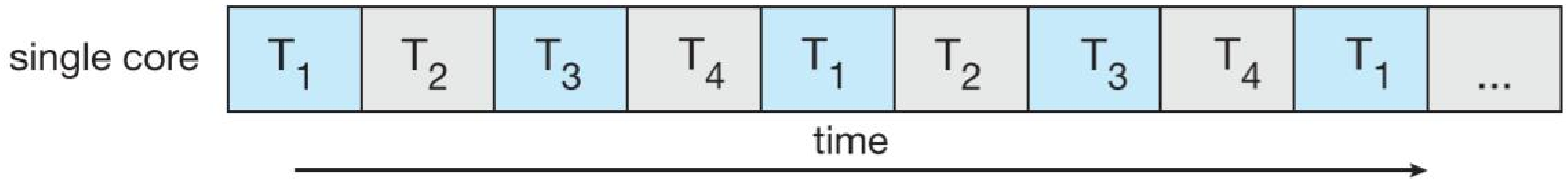


Figure 4.3 Concurrent execution on a single-core system.

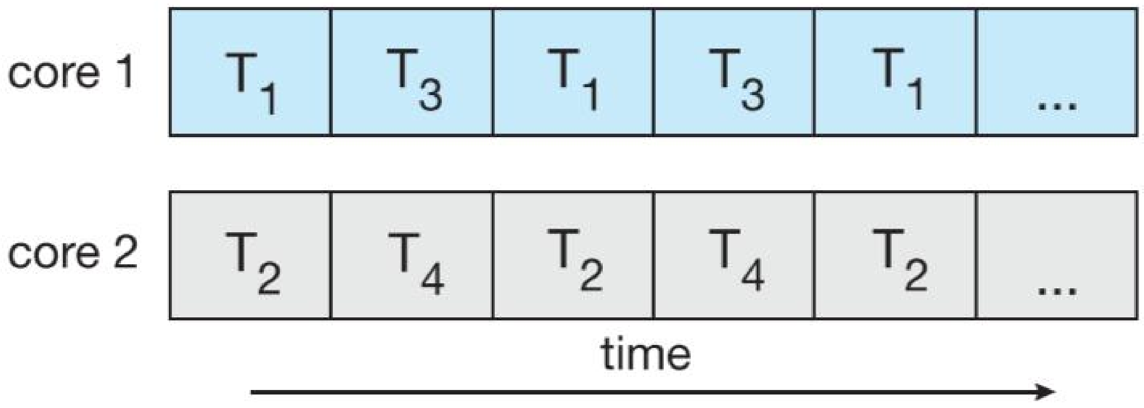


Figure 4.4 Parallel execution on a multicore system.

User Threads

- Thread management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads (can also be provided as system library)
  - Win32 threads
  - Java threads

Kernel Threads

- Supported by the Kernel

- Almost all contemporary OS implements kernel threads. Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - MacOS

Multithreading Models

- **Many-to-One**
  thread management is efficient, but will block if making system call, kernel can schedule only one thread at a time
- **One-to-One**
  more concurrency, but creating thread is expensive
- **Many-to-Many**

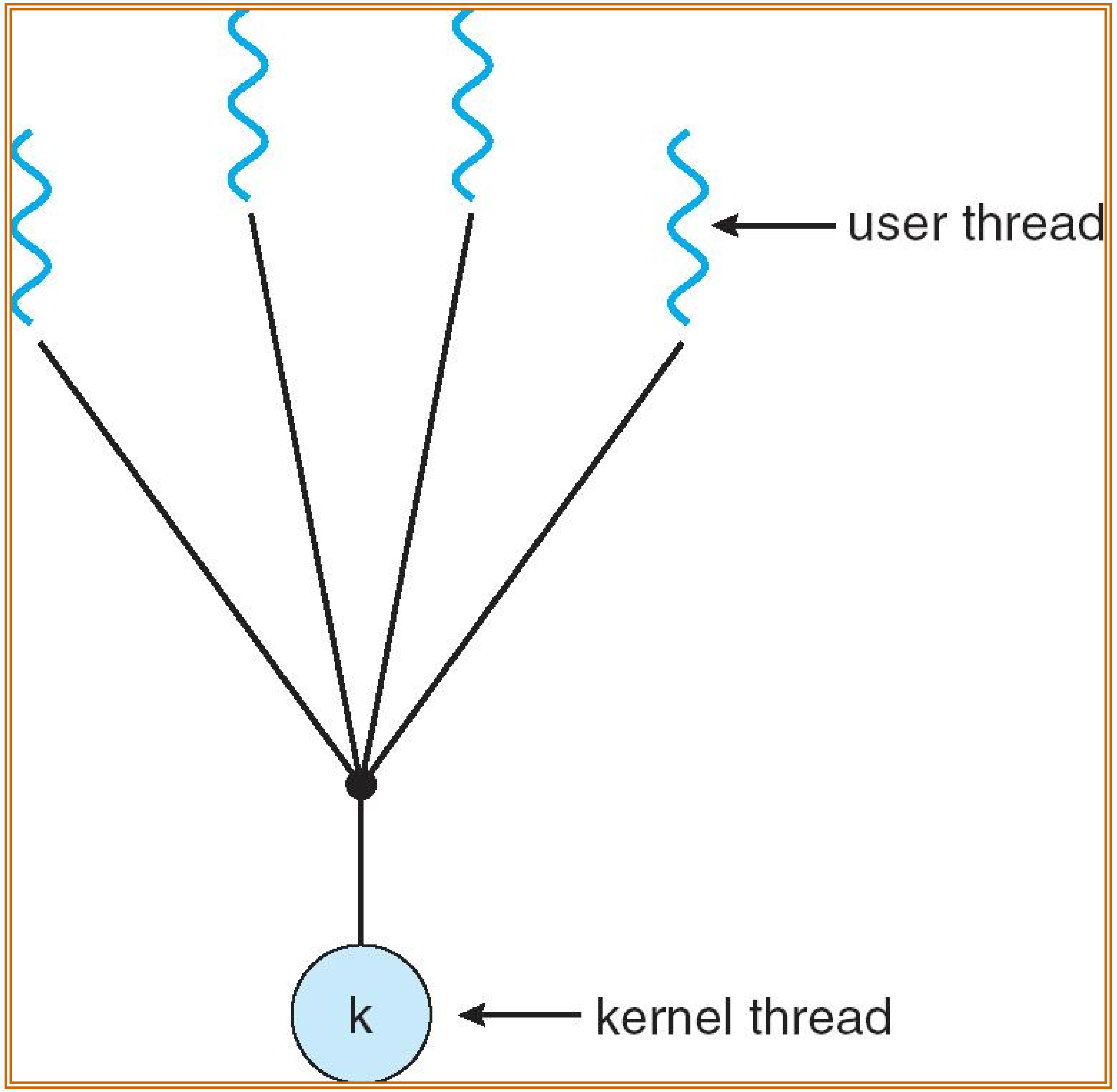
  flexible

Many-to-One

- Many user-level threads mapped to single kernel thread
- The scheduling is done completely by the thread library and the kernel itself is not aware of the multiple threads in user-space.
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

Many-to-One Model

线程的分配，管理，调度在线程库的层次实现，实际上只对应一个kernel thread (local scheduling)

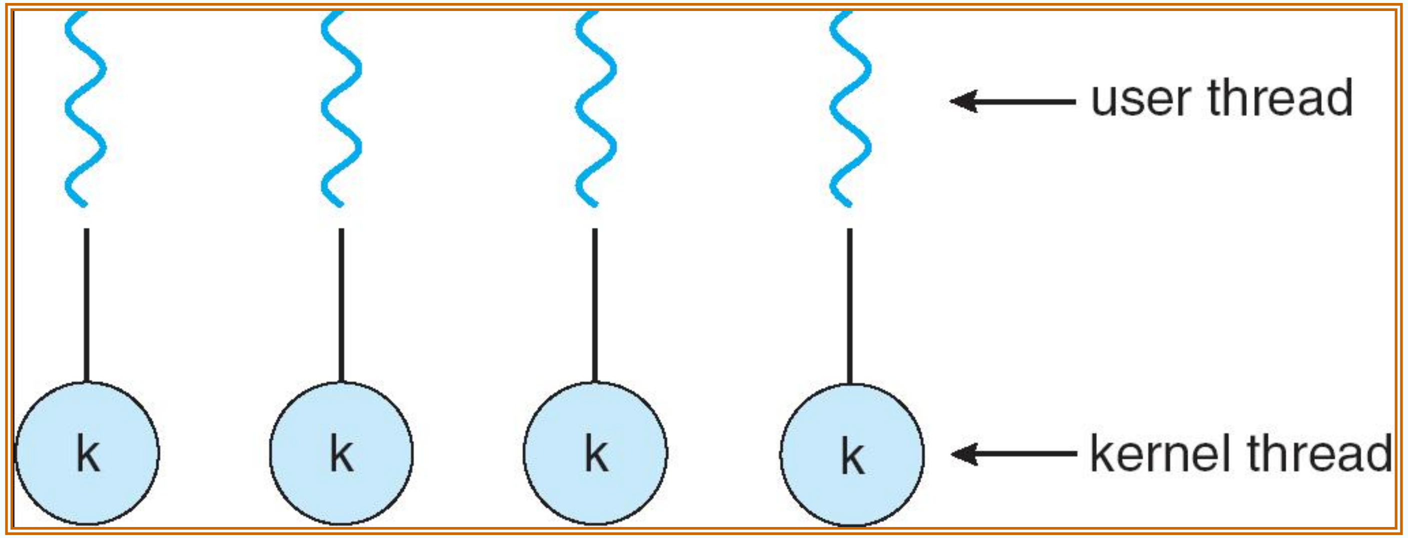


One-to-One

- Each user-level thread maps to kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

线程的分配,管理,调度在kernel的层次实现,由操作系统全权负责(global scheduling)

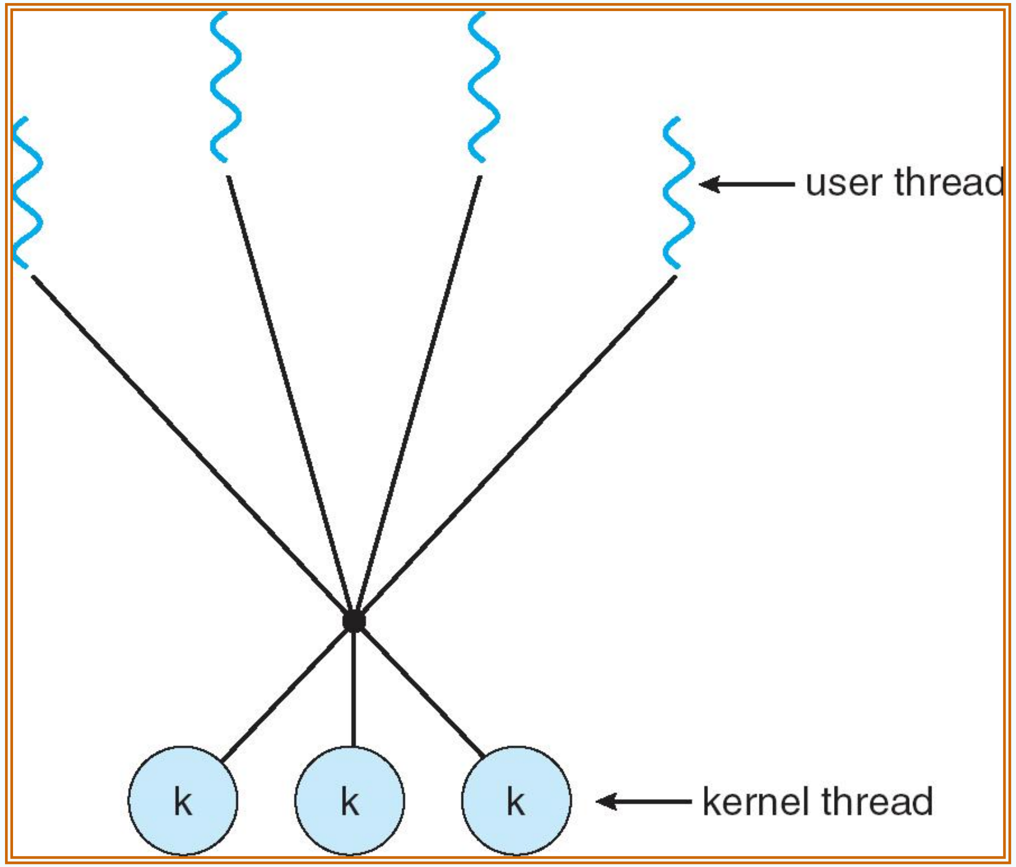
One-to-one Model



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
- A program can have as many threads as are appropriate without making the process too heavy or burdensome. In this model, a user-level threads library provides sophisticated scheduling of user-level threads above kernel threads.
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package
- Go routines

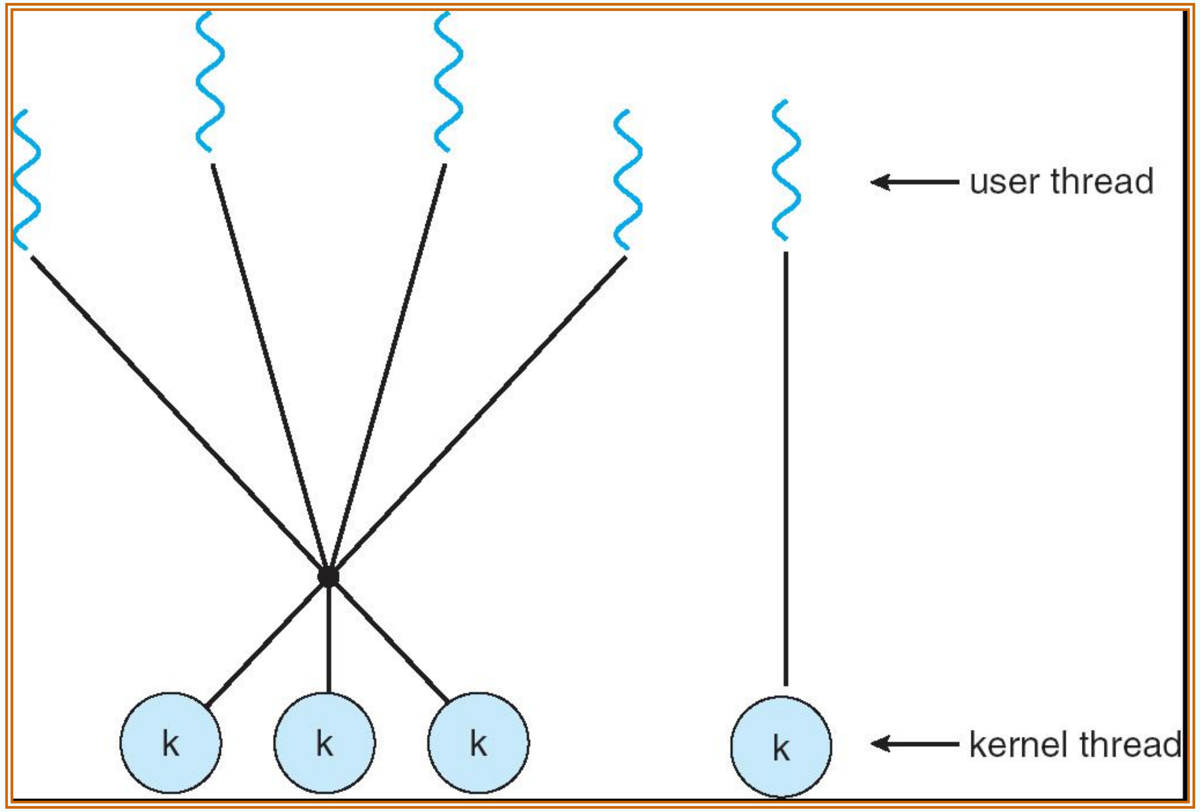
Many-to-Many Model



Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to a kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

Two-level Model



Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Thread cancellation
- Signal handling
- Thread pools
- Thread-specific data
- Scheduler activations

Semantics of fork() and exec()

- Does **fork()** duplicate only the calling thread or all threads?
- Some Unix systems have two versions of fork(), one that duplicates all threads and another that duplicates the thread that invokes fork(). It’s not trivial though.
- Exec() will replace the entire process.

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, either synchronous or asynchronous:
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal must be handled
- Options: (method of delivery depends on the type of signal)
  - 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 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 via a flag if it should be cancelled

```
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);
```

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

- Thread-Local Storage (in 10th edition)
- Allows each thread to have its own copy of data
- In some ways similar to static data, but are unique to each thread
- 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
- LWP is a **virtual processor** attached to kernel thread
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the thread library
- Upcalls are handled by the thread library with an **upcall handler**
- This communication allows an application to maintain the correct number of kernel threads

  when an application thread is about to block, an upcall is triggered.

Pthreads

Q: Is it a user- or kernel-level library?

- 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)

Can be either user- or kernel-level.

Windows XP Threads

- Implements the one-to-one mapping
- 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

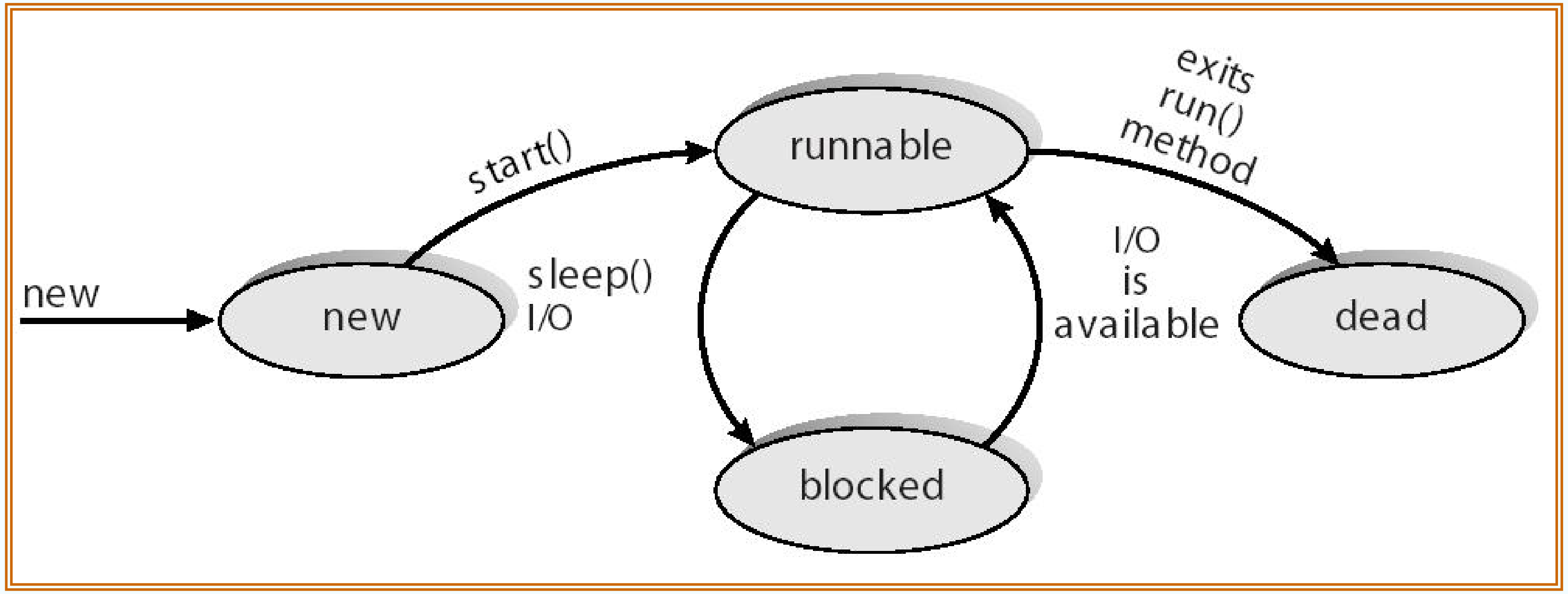
- Linux refers to them as *tasks* rather than *threads*
- Thread creation is done through **clone()** system call
- **clone()** allows a child task to share the address space of the parent task (process)

Java Threads

- Java threads are managed by the JVM

- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

Java Thread States



End of Chapter 4