The picture can't be displayed.

# 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. And still not responsive!

# To make it responsive enough, we need to

- Break the operations into very very small pieces;
- However, to be efficient enough, we want to execute 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!


Silberschatz, Galvin and Gagne ©2005

# Multi-thread version of the program

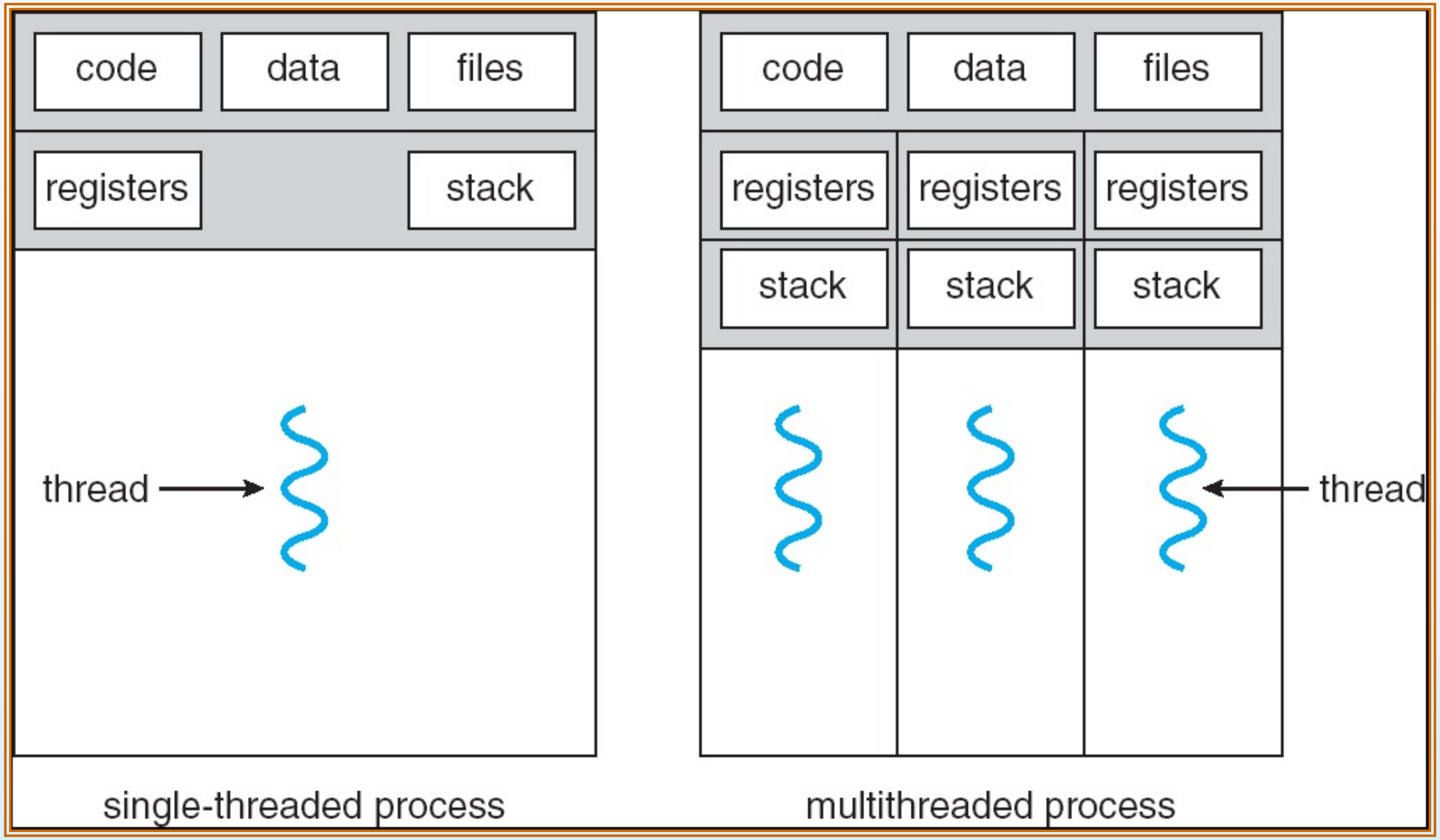
```
Main(){
        CreateThread (RetrieveData());
        CreateThread (DisplayData());
        CreateThread (GetInputEvents());
        WaitForThreads();
}

// 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.

# Thread VS Process

- processes are typically independent, while threads exist as subsets of a process
- processes carry considerably more state information than threads, whereas multiple threads within a process share process state as well as memory and other resources
- processes have separate address spaces, whereas threads share their address space
- processes interact only through system-provided inter-process communication mechanisms
- context switching between threads in the same process is typically faster than context switching between processes.

# 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


Silberschatz, Galvin and Gagne ©2005

# Kernel Threads

- Supported by the Kernel
- Almost all contemporary OS implements kernel threads. Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

# Multithreading Models

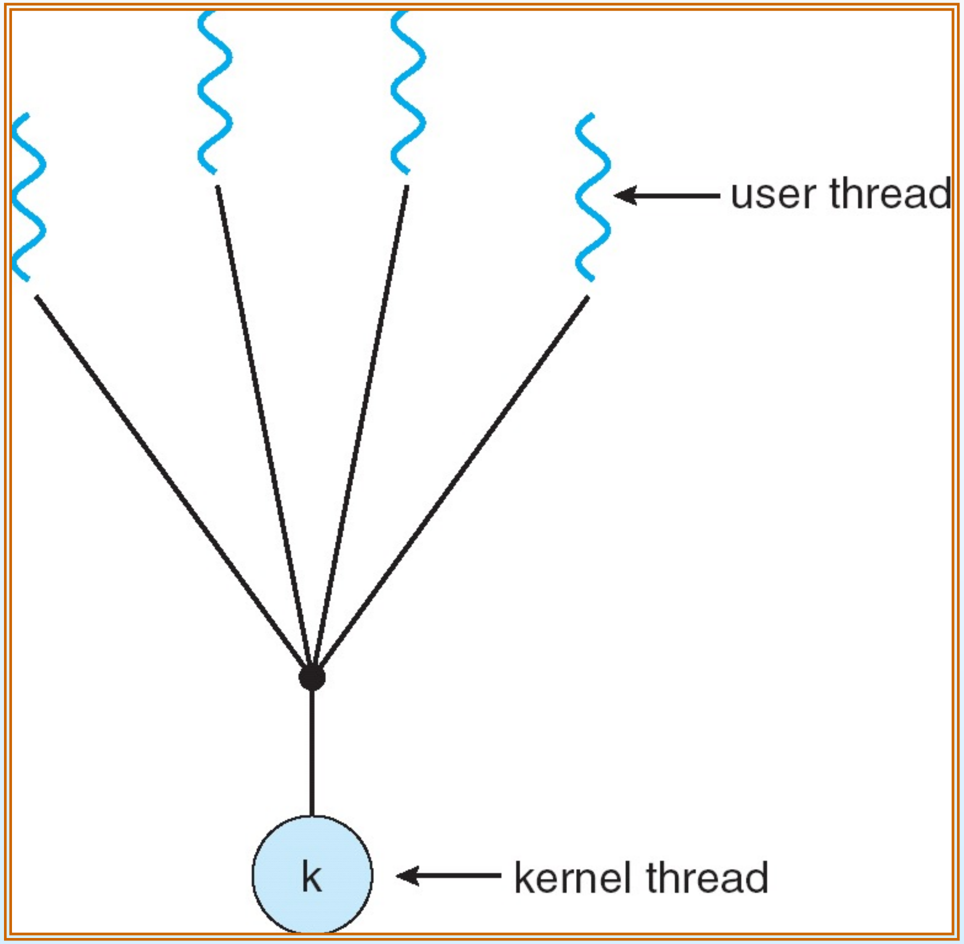
- Many-to-One
  thread mgmt is efficient, but will block if making system call, kernel can schedule only one thread at a time
- One-to-One
  more concurrency, but creating thread is expensive
- Many-to-Many
  flexible

# Many-to-One

- Many user-level threads mapped to single kernel thread
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads


Silberschatz, Galvin and Gagne ©2005

# Many-to-One Model

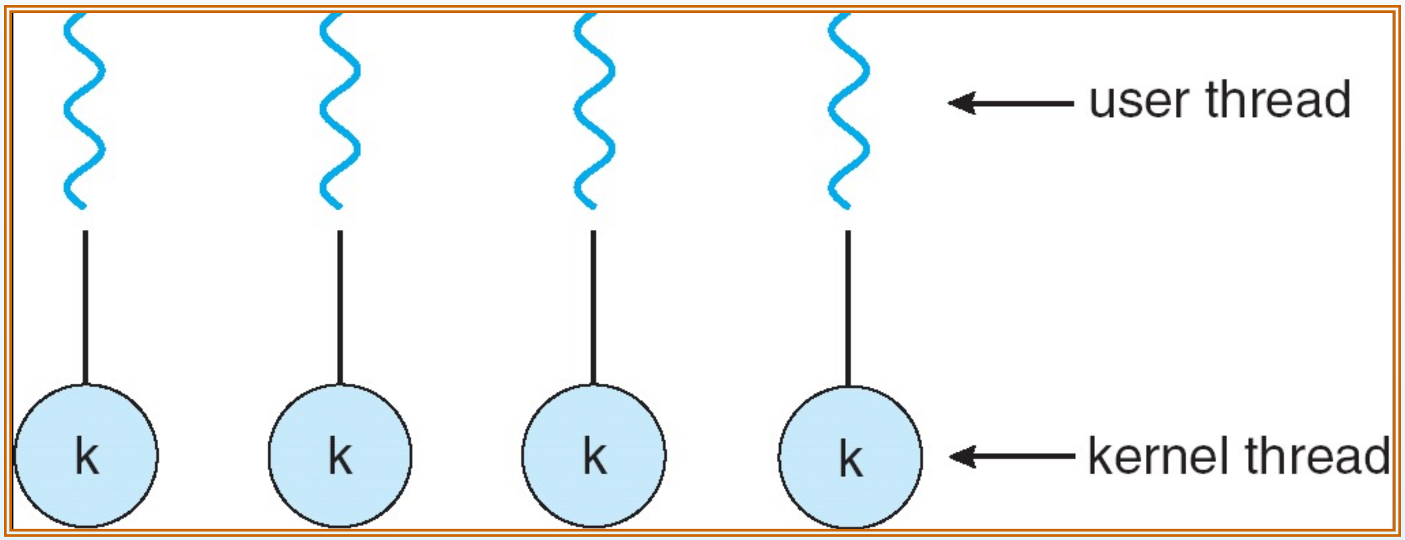


# One-to-One

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


Silberschatz, Galvin and Gagne ©2005

# One-to-one Model

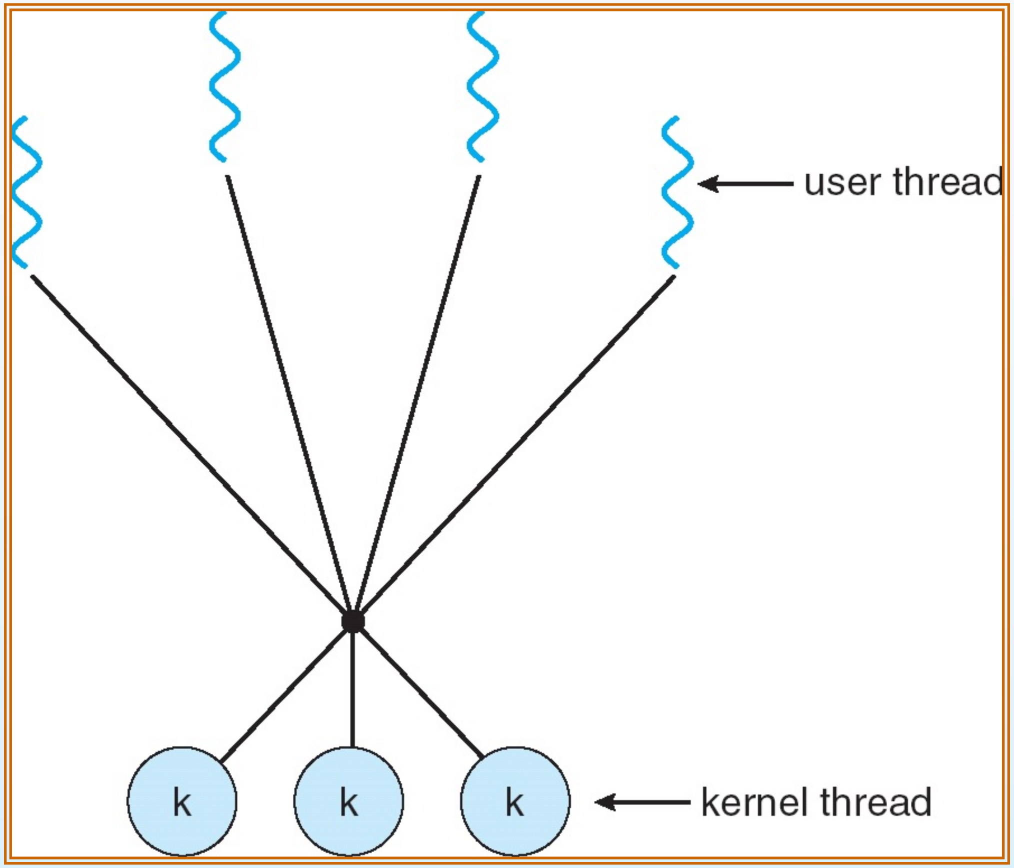


# Many-to-Many Model

- 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


Silberschatz, Galvin and Gagne ©2005

# Many-to-Many Model

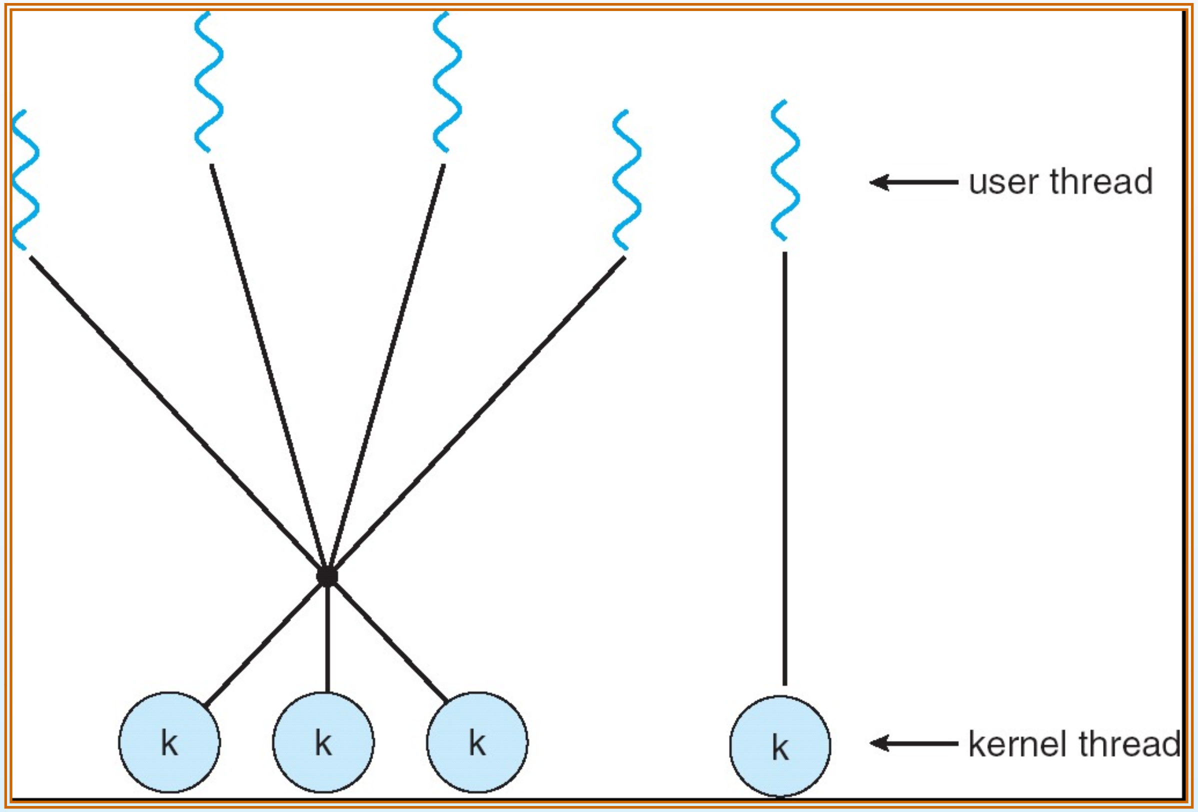


# Two-level Model

- 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


Silberschatz, Galvin and Gagne ©2005

# Two-level Model



# Threading Issues

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


Silberschatz, Galvin and Gagne ©2005

# 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().
- Exec() will replace the entire process.


Silberschatz, Galvin and Gagne ©2005

# Thread Cancellation

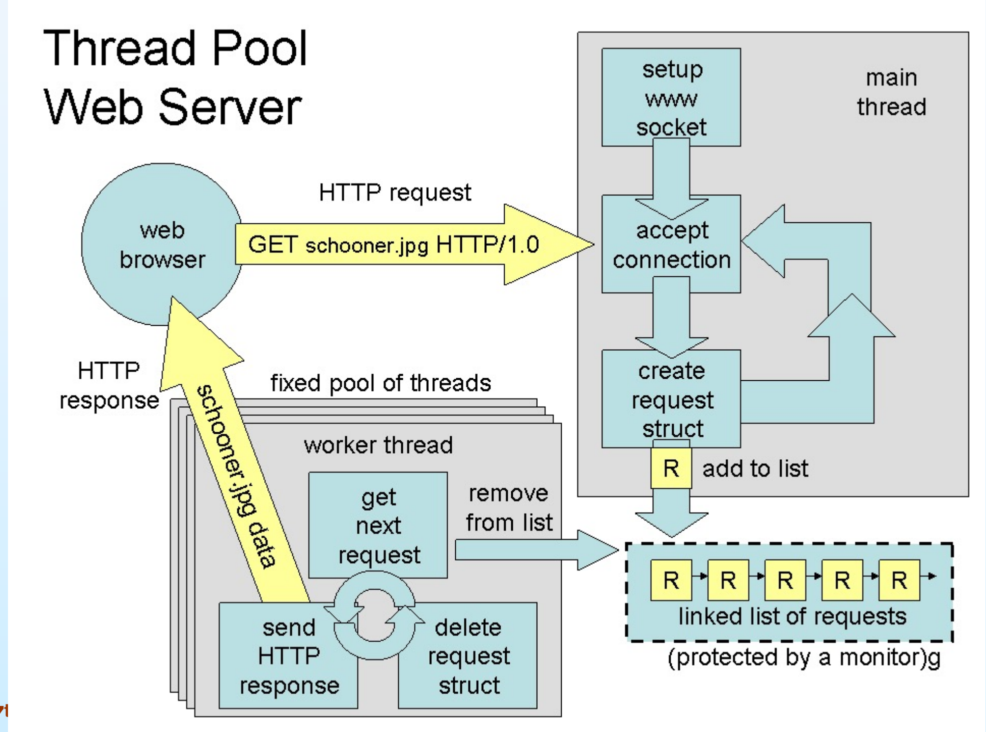
- Terminating a thread before it has finished
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check via a flag if it should be cancelled

# 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: (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 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

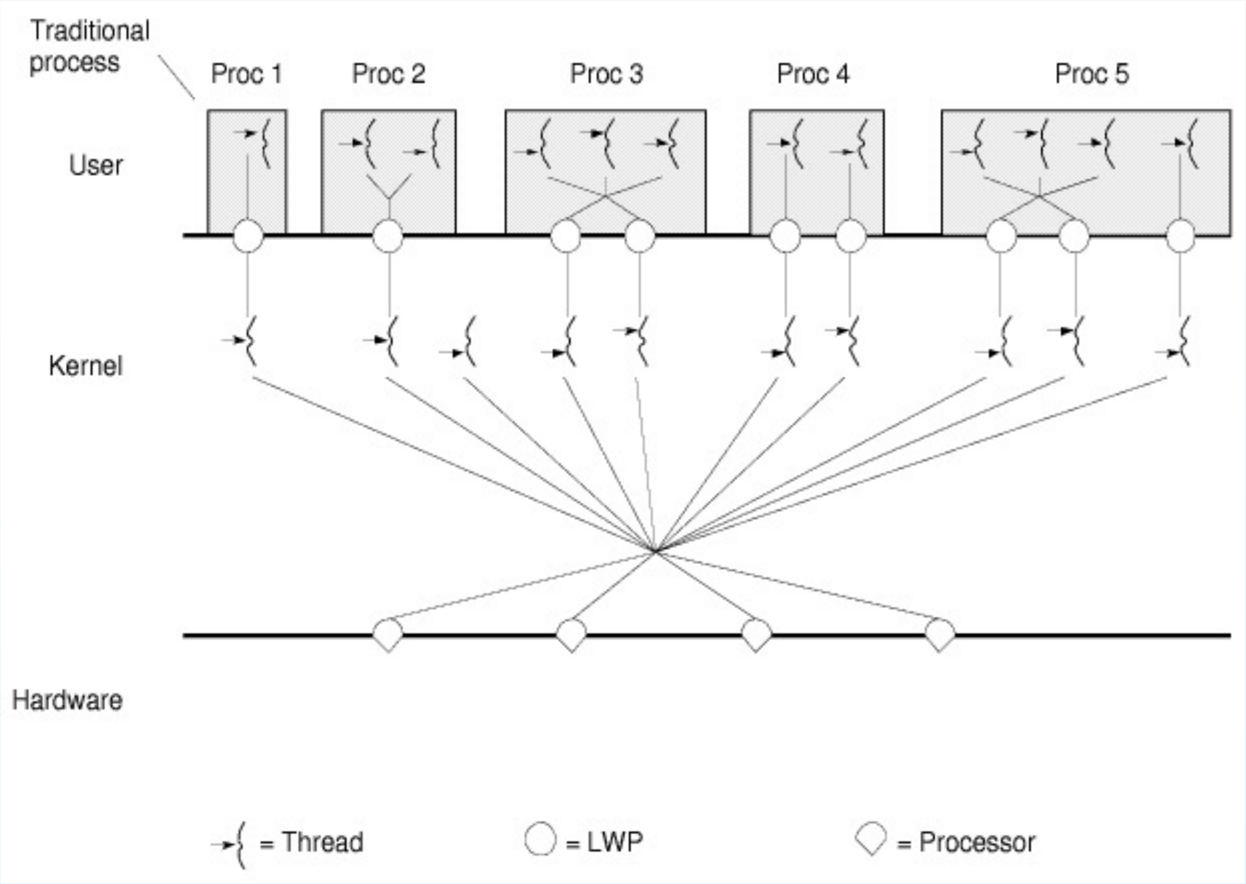
- Allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
  - Unlike processes, all threads in a single program share the same address space. This means that if one thread modifies a location in memory (for instance, a global variable), the change is visible to all other threads.
  - **thread-specific data area:** The variables stored in this area are duplicated for each thread, and each thread may modify its copy of a variable without affecting other threads.

# Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- LWP (**Light-weight process**) 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.*

# LWP : Light-weight Process



# 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)

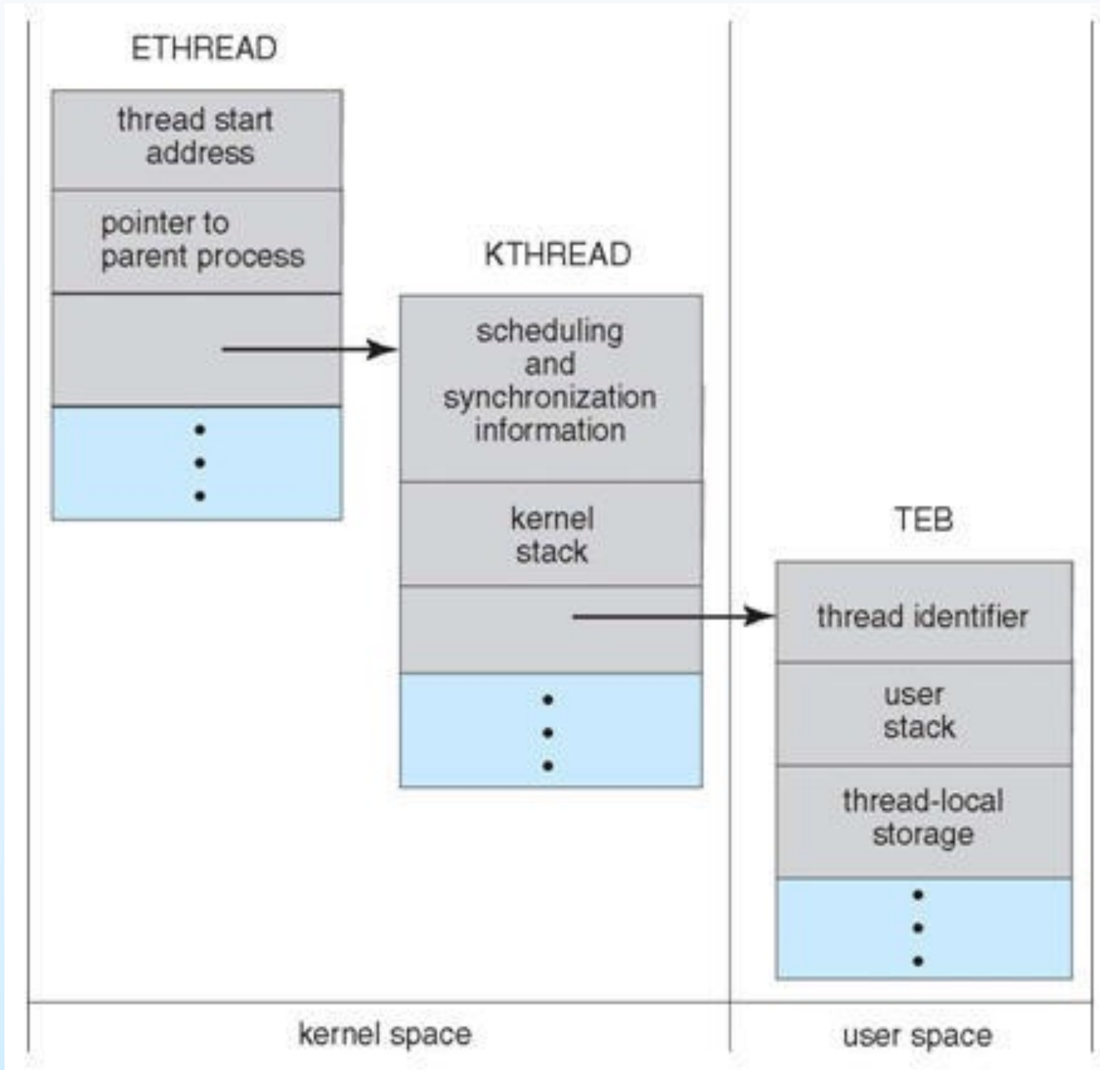
Can be either user- or kernel-level.

# 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)


Silberschatz, Galvin and Gagne ©2005

# Structures of Thread



# Linux Threads

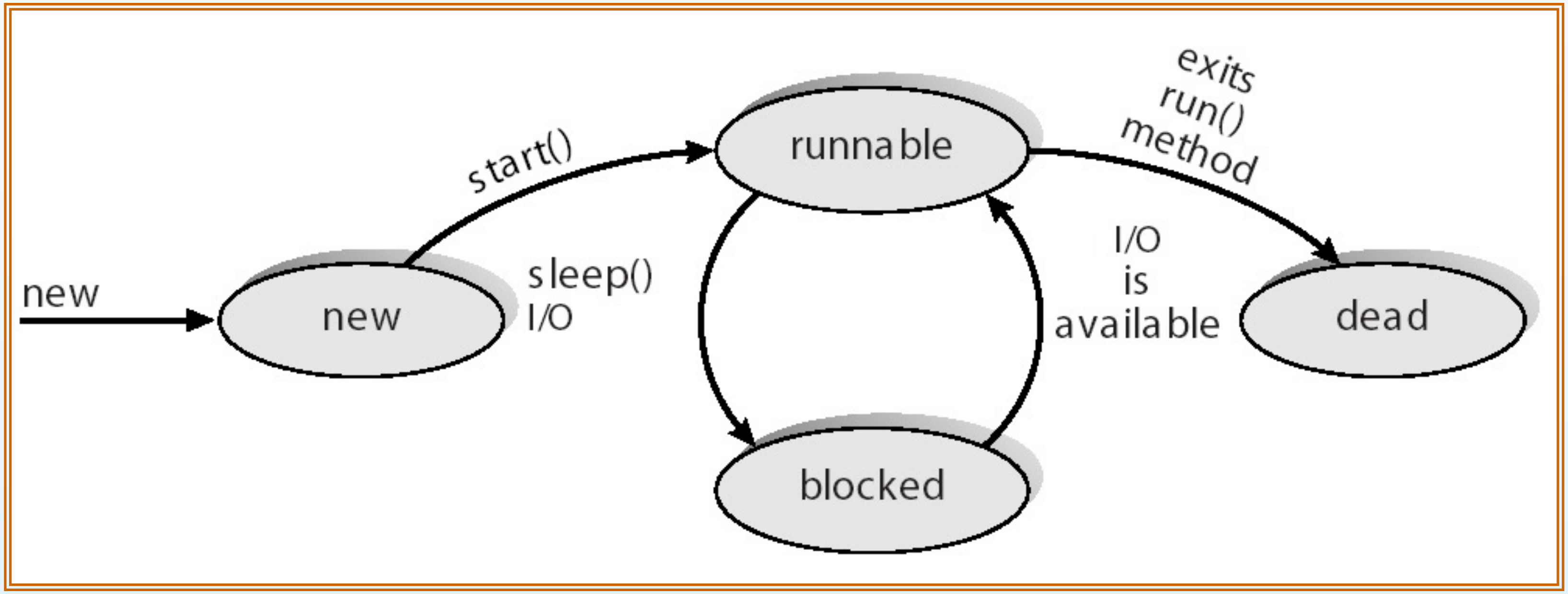
- Linux refers to them as *tasks* rather than *threads*
- Thread creation is done through **clone()** system call
- **clone()** allows a child task to share the address space of the parent task (process)

# Java Threads

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


Silberschatz, Galvin and Gagne ©2005

# Java Thread States



The picture can't be displayed.

# End of Chapter 4