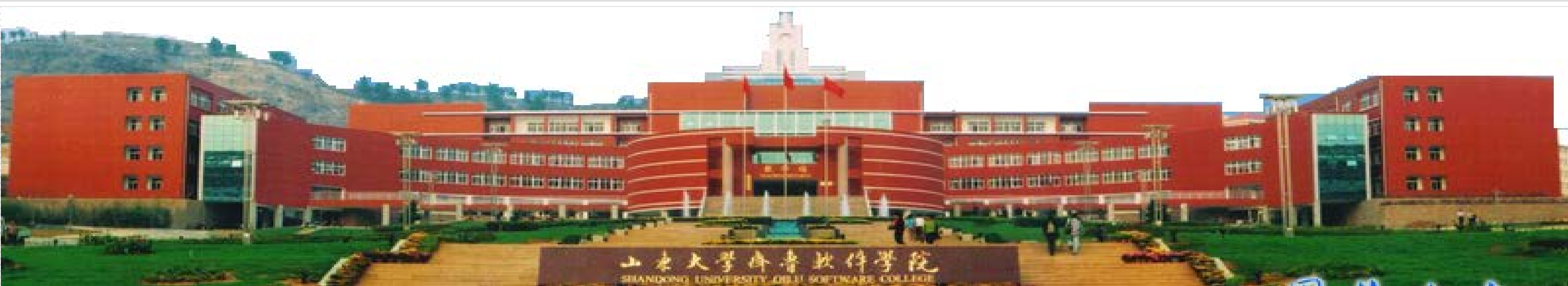
Chapter 4

Threads



Contents

Objectives

- To introduce the notion of a thread---a fundamental unit of CPU utilization that forms the basis of multithread computer system.
- To discuss the APIs for Pthreads, Win32, and Java thread libraries.

Thread

- A thread
  - A running entity of a process, and a unit that can be scheduled independently.
  - A basic unit of CPU utilization

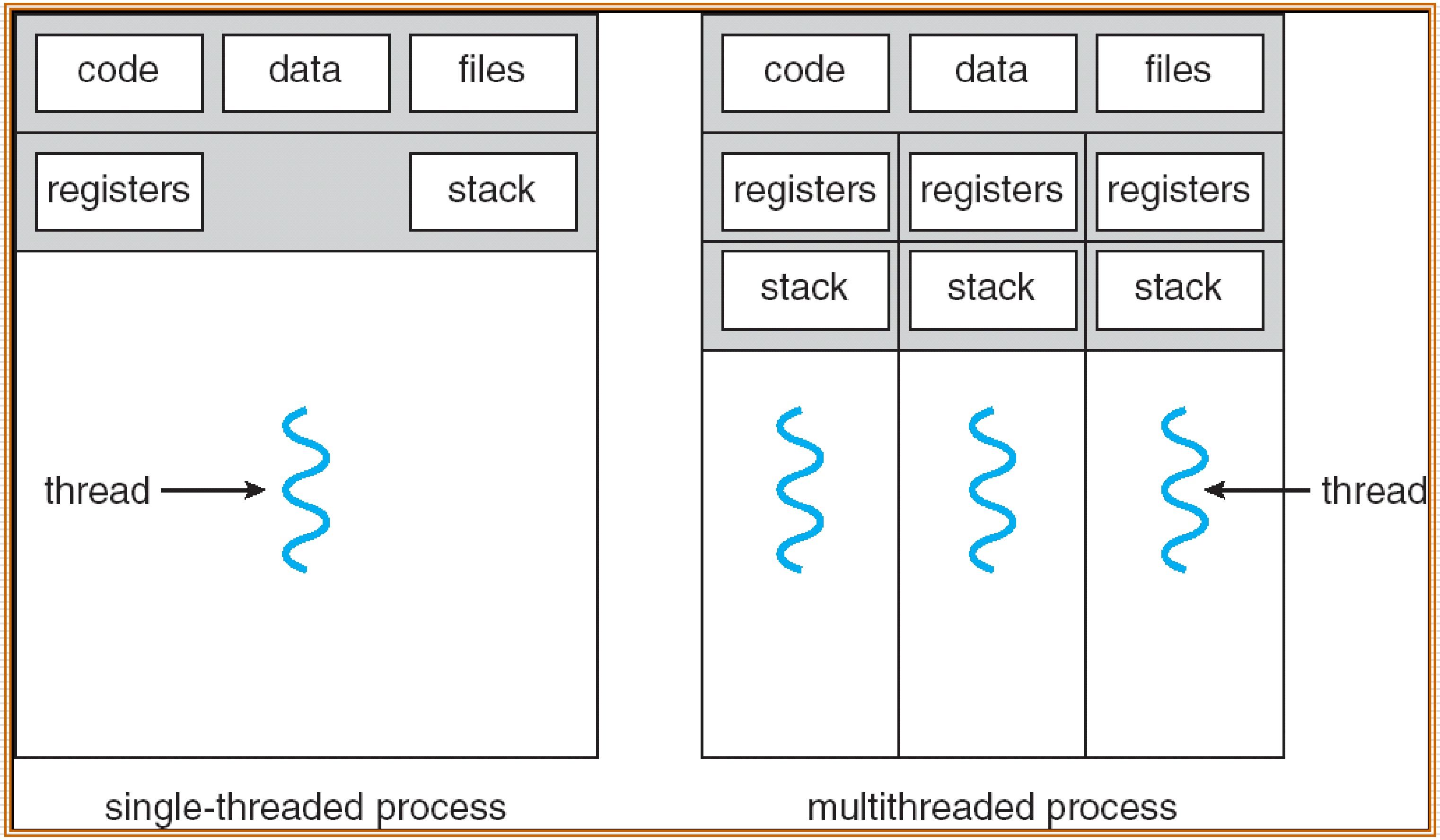
Motivation

- When increase the concurrence of system, the time spent on process creation, process cancellation, process exchange will increase greatly
- In addition, the communication between processes is also limited.

Motivation --- example

- Suppose there is a web server
  - What is the result if there is only one thread?
  - The time to create
  - The time to exchange
  - The space for each user
- A program will accept input from user, list the menu, execute the command
  - What is the result if there is only one thread?

Single and Multithreaded Processes



Benefits

- Responsiveness
- Resource Sharing
- Economy
- Utilization of MP Architectures

Thread

- A thread
  - A running entity of a process, and a unit that can be scheduled independently.
  - A basic unit of CPU utilization
- Resources still belong to process
  - Code section
  - Data section
  - Open files
  - Signals

Thread & Process

- Process is the owner of resources
  - Code section
  - Data section
  - Open files
  - Signals
- Thread is a running unit (smallest unit)
  - Thread has few resources (counter, register, stack), shares all the resources that the process has.
- A program has one process at least, and one process has one thread at least

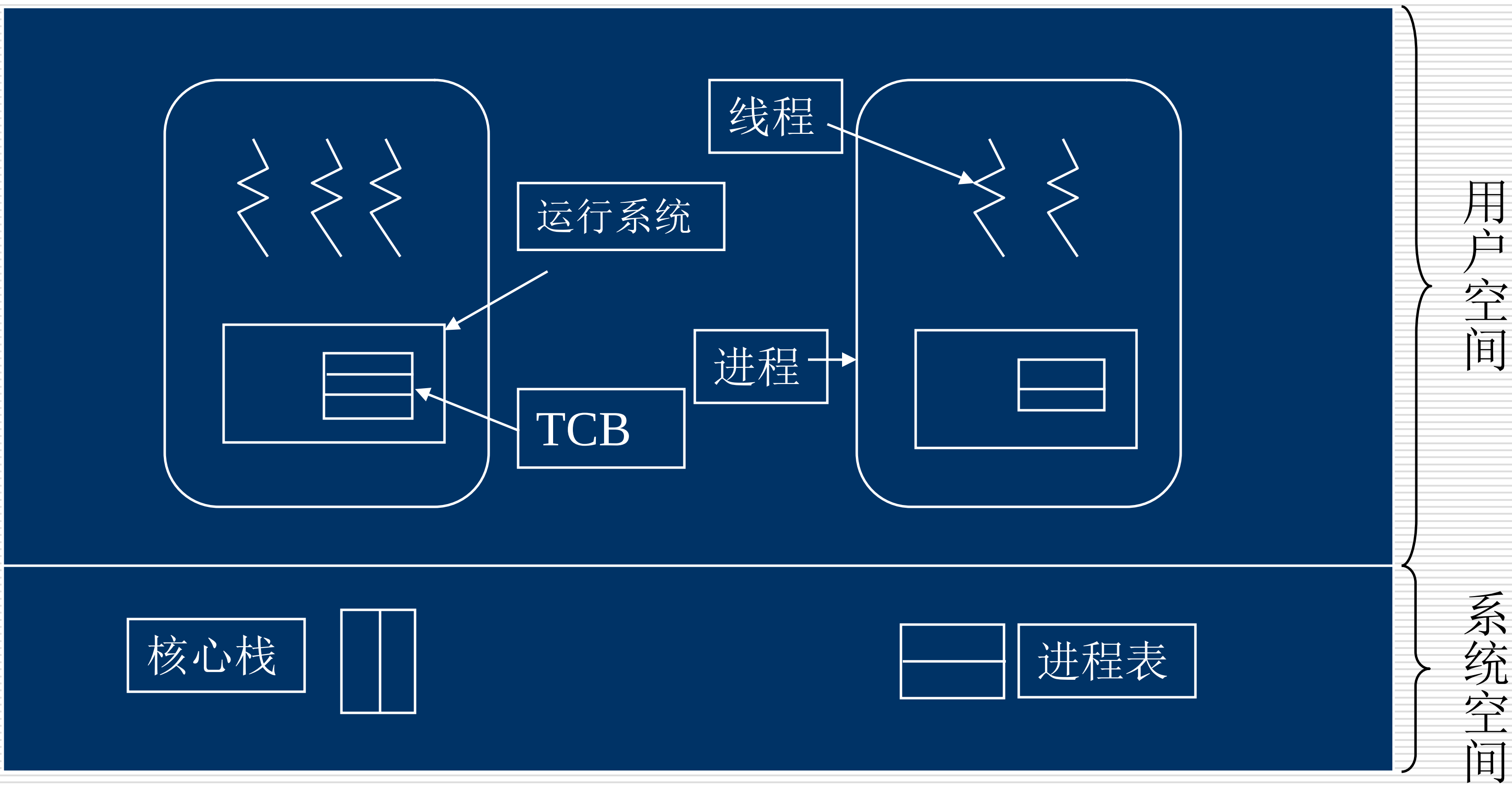
Implementation

- User Level Thread
- Kernel level thread
- Hybrid method

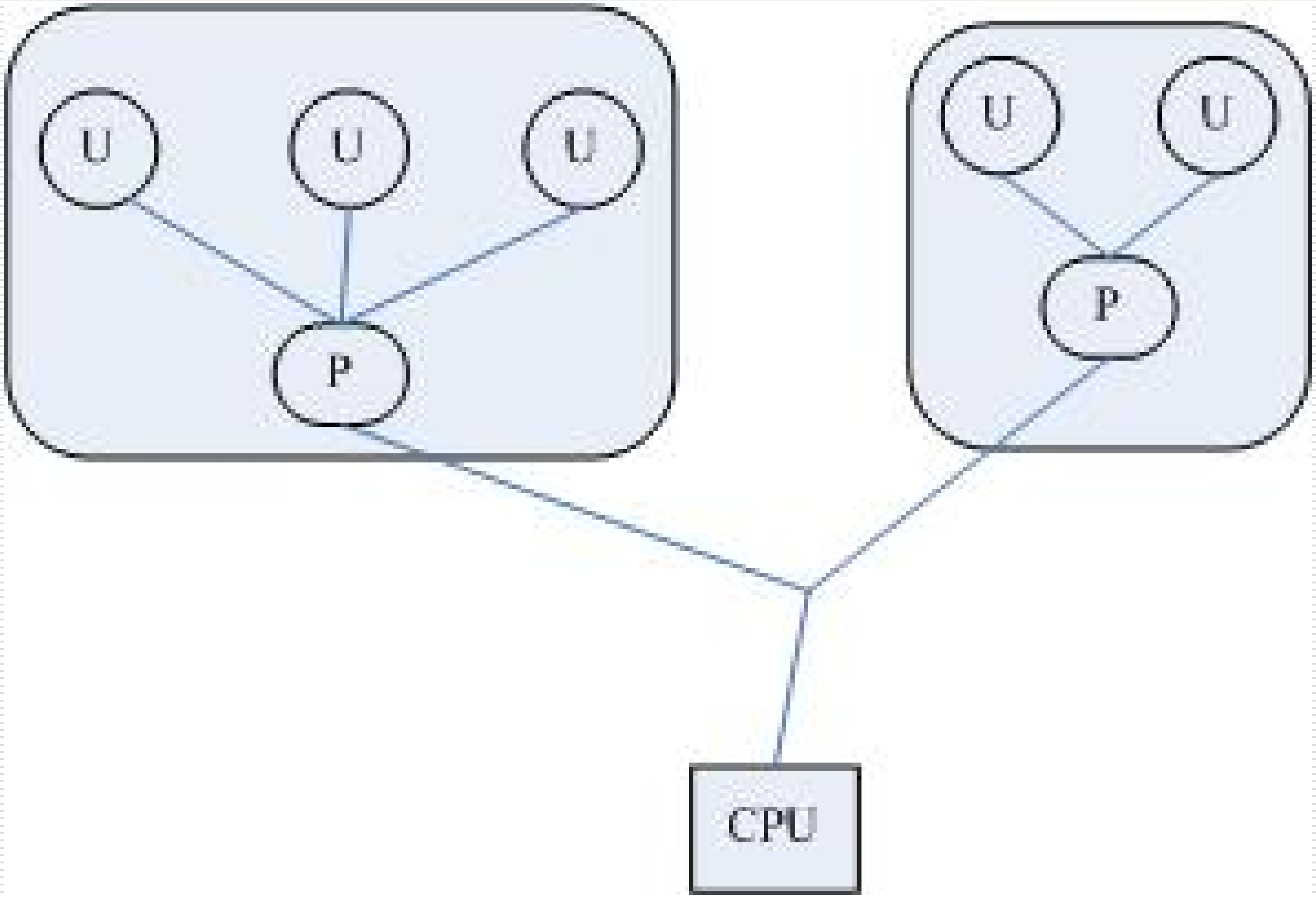
User Threads

- Thread management done by user-level threads library
- Kernel knows nothing about threads

User thread



User thread



User thread

- Implemented by thread library
  - Create, cancellation
  - Transfer data or message
  - Save and recover the context of threads
- The kernel manage the process, but know nothing about thread
- When a thread have a system call, the process will be blocked. To thread library, the thread’s state is running

User thread

- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

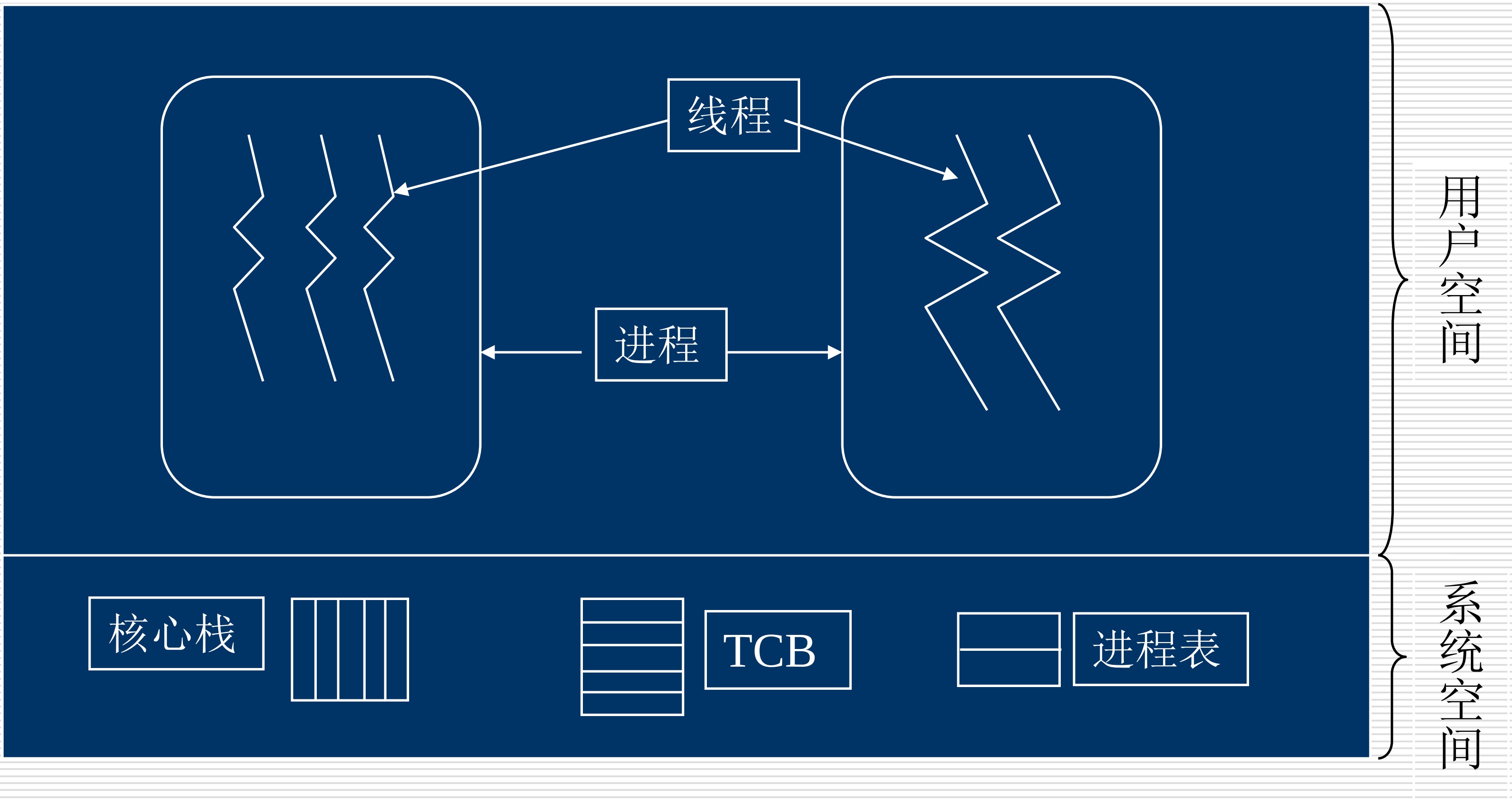
Advantages & Disadvantages

- Advantages
  - It does not need to call the kernel when there is thread switching.
  - Scheduling is determined by application, so best algorithm can be selected.
  - ULT can run on any platform if the thread library is install on it.
- Disadvantages
  - Most system call will result in blocking
  - Two threads in the same process can not simultaneously run on two processors

Kernel Threads

- Supported by the Kernel
- All threads are managed by the kernel
  - Create, cancellation and schedule
  - No thread library, but provide API
  - Kernel maintains context of threads and processes
  - The switch between threads needs the support of kernel
- Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Kernel Threads



Advantages & Disadvantages

- Advantages
  - For multiprocessor system, more than one thread can run simultaneously
  - Just block the thread, not process
- Disadvantage
  - The switch between threads in the same process, will slow the speed.

Hybrid model

- Thread is created in user space

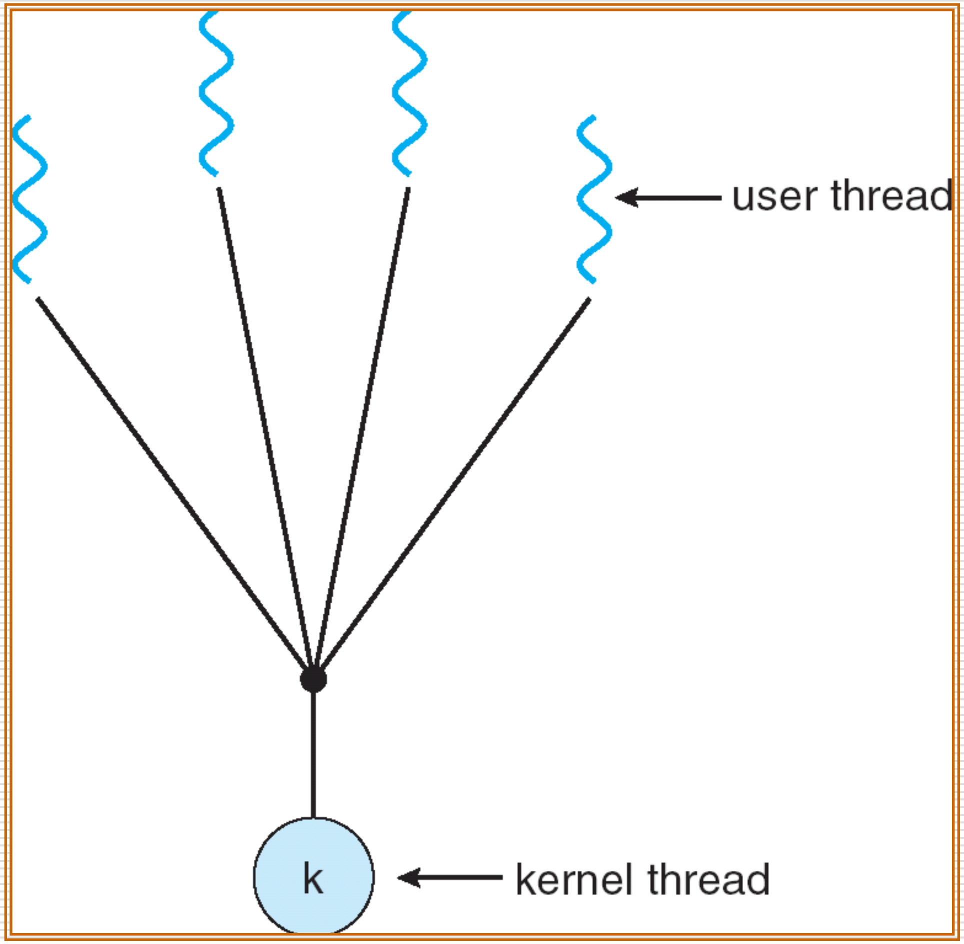
Multithreading Models

- 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
- Advantage
  - Management is efficient
- Disadvantages
  - Process is blocked when one thread is blocked
  - Can’t utilize multi-processors system

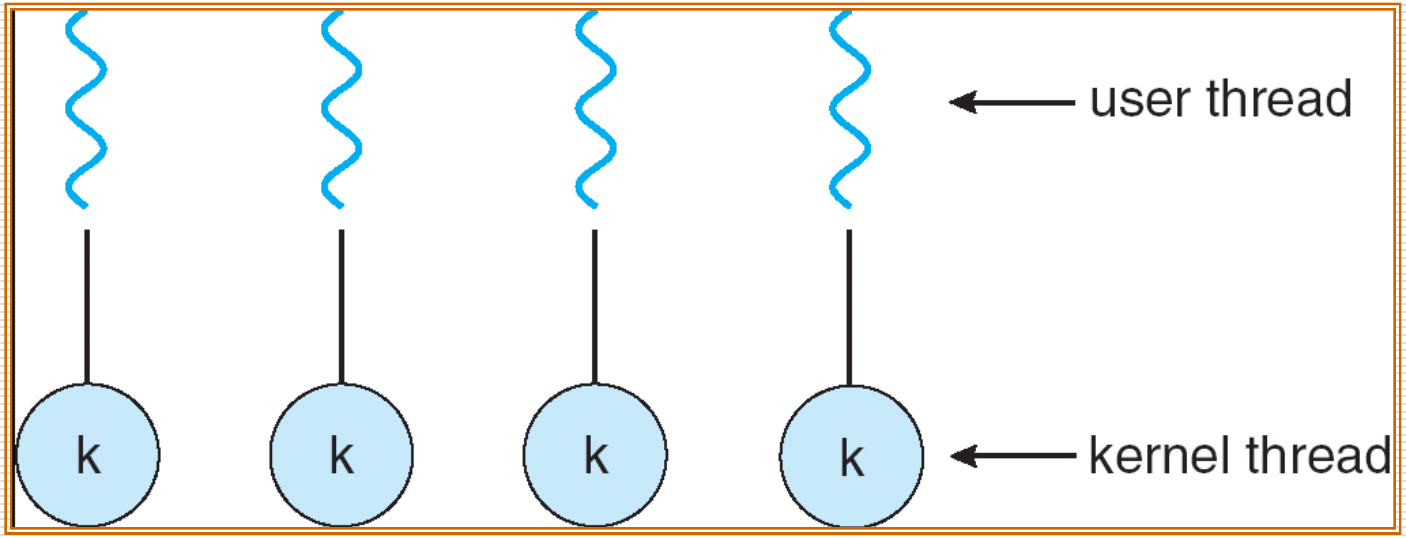
Many-to-One Model



One-to-One

- Each user-level thread maps to one kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later
- Advantage
  - Can run on multiprocessor system
  - One blocked, others can run still
- Disadvantage
  - To create one user thread, one kernel thread is also created.

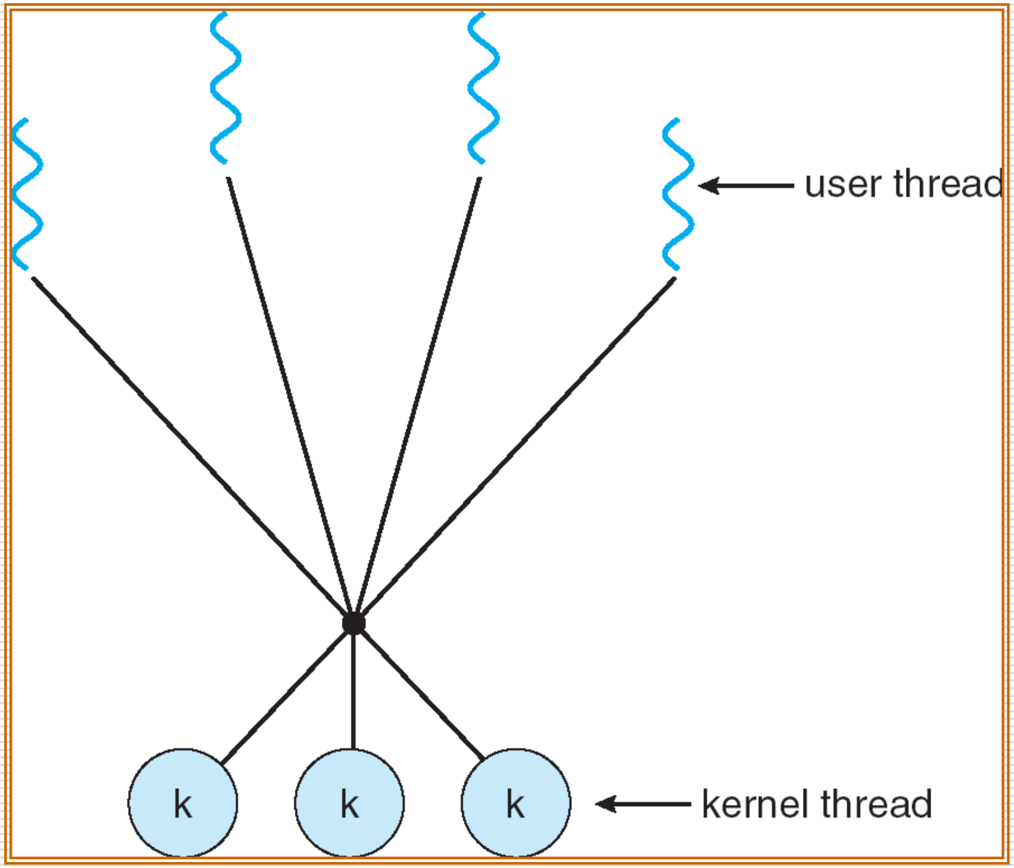
One-to-one Model



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

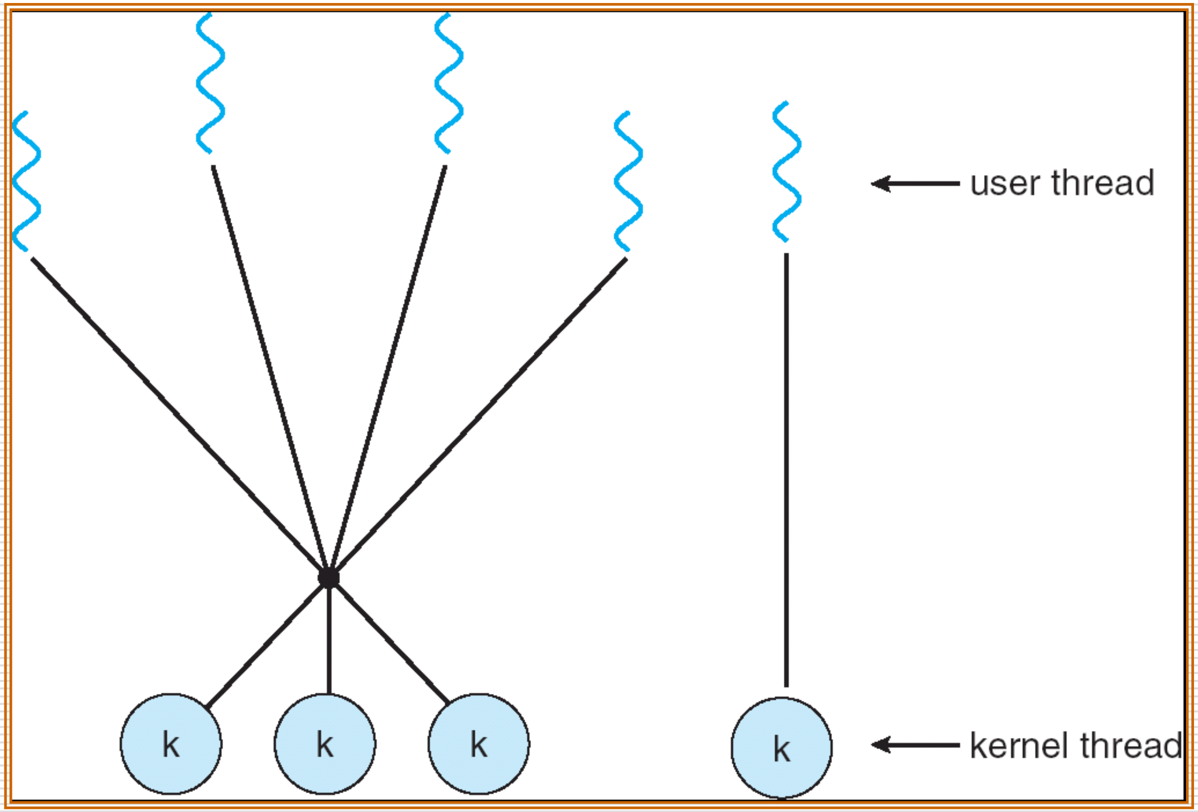
Many-to-Many Model



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



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?
  - Exec() is after fork()
  - No exec() after fork()

Thread Cancellation

- Terminating a thread before it has finished
  - Search database
  - Web pages
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
    - Cancellation point

Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
  - Synchronous
  - Asynchronous
- 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

- Problems in multithread server:
  - Spend much time to create thread
  - Resources will be exhausted if no limitation to thread
- 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
  - Example—transaction processing system

Scheduler Activations

- Both N: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

Pthreads

- 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

```
#include<pthread.h>
#include<stdio.h>
int sum; /*this data is shared by the thread(s) */
void *runner(void *param); /*the thread*/

Main(int argc, char *argv[])
{
        pthread_t tid; /*the thread identifier*/
        pthread_attr_t attr; /* set of attributes for the thread*/
        pthread_attr_init(&attr);
        pthread_create(&tid, &attr, runner, argv[1]);
        pthread_join(tid, NULL);
        printf("sum= %d\n", sum);
}

void *runner(void *param)
{
        int upper = atoi(param);
        int I;
        sum = 0;
        if (upper > 0) {
                for (I = 1; I <= upper; I ++)
                        sum += I;
        }
        pthread_exit(0);
}
```

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

Extending Thread class

```
class Worker1 extends Thread
{
    public void run() {
        System.out.println("I Am a Worker Thread");
    }
}

public class First
{
    public static void main(String args[]) {
        Worker1 runner = new Worker1();
        runner.start();

        System.out.println("I Am The Main Thread");
    }
}
```

Runnable interface

```
public interface Runnable
{
    public abstract void run();
}
```

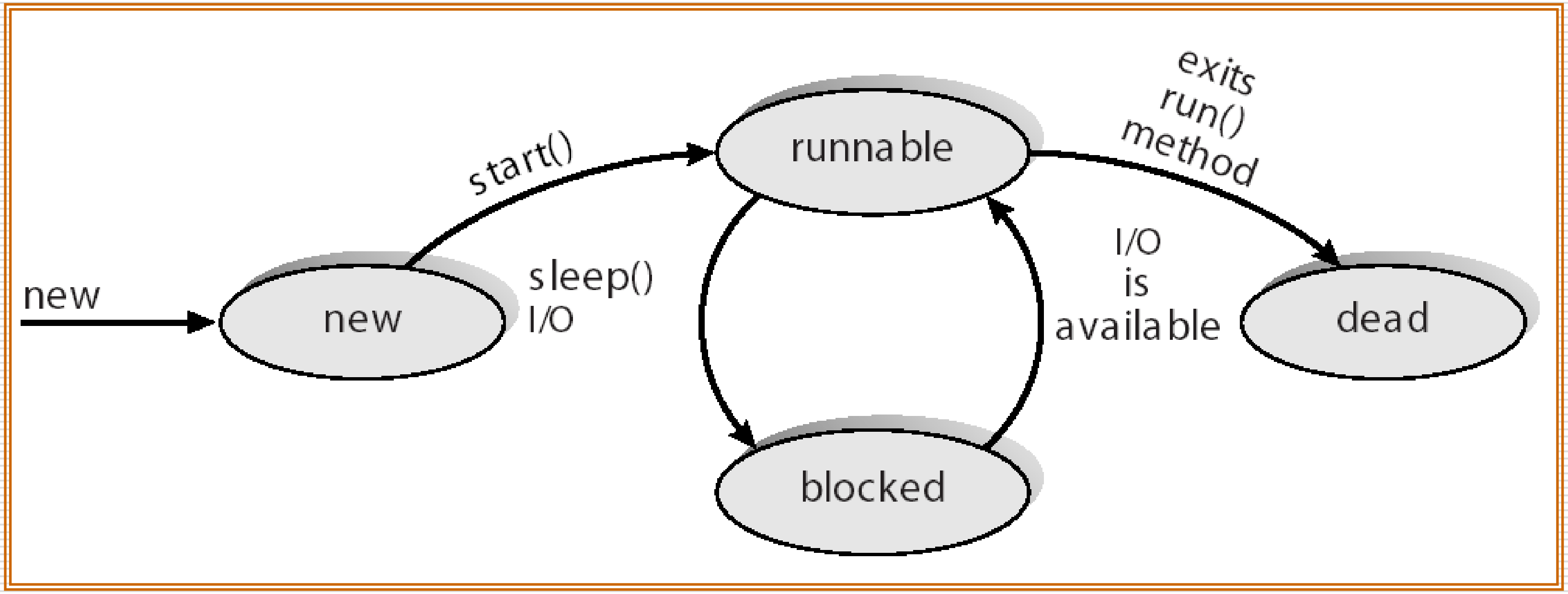
Implementing the Runnable interface

```
class Worker2 implements Runnable {
    public void run() {
        System.out.println("I Am a Worker Thread");
    }
}

public class Second {
    public static void main(String argc[]) {
        Runnable runner = new Worker2();
        Thread thrd = new Thread(runner);
        thrd.start();

        System.out.println("I Am The Main Thread");
    }
}
```

Java Thread States



Joining Threads

```
class JoinableWorker implements Runnable {
    public void run() {
        System.out.println("Worker working");
    }
}

public class JoinExample {
    public static void main(String [] args) {
        Thread task = new Thread(new JoinableWorker());
        task.start();

        try { task.join(); }
        catch (InterruptedException ie) { }
        System.out.println("Worker done");
    }
}
```

Thread cancellation

```
Thread thrd = new Thread(new
    InterruptibleThread());
thrd.start();

…

//now interrupt it
thrd.interrupt();
```

Thread cancellation

```
public class InterruptibleThread implements Runnable {
    public void run() {
        while (true) {
            ...
            if (Thread.currentThread().isInterrupted())
    {
                System.out.println();
                break;
            }  /* 线程取消点 */
        }
        //clean up and terminate
    }
}
```

Thread data

```
Class Service {
	private static ThreadLocal errorCode = new ThreadLocal();

	public static void transaction() {
		try {
		}
		catch (Exception e) {
				errorCode.set(e);
		}
	}

	public static Object getErrorCode() {
		return errorCode.get();
	}
}
```

```
class Worker implements Runnable {
    private static Service provider; //线程特定数据
    public void run() {
        provider.transaction();
        System.out.println(provider.getErrorCode());
    }
}
```

Producer-consumer problem

```
public class Factory {
    public Factory() {
        Channel mailBox = new MessageQueue();
        Thread producerThread = new Thread(new Producer(mailBox));
        Thread consumerThread = new Thread(new
    Consumer(mailBox));
        producerThread.start();
        consumerThread.start();
    }

    public static void main(String args[]) {
        Factory server = new Factory();
    }
}
```

Producer thread

```
class Producer implements Runnable {
	private Channel mbox;

	public Producer(Channel mbox) {
		this.mbox = mbox;
	}

	public void run() {
		Date message;
		while (true) {
				SleepUtilities.nap(); //小睡片刻
				message = new Date();
				System.out.println(“Producer produced” + message);
				mbox.send(message);
		}
	}
}
```

Consumer thread

```
class Consumer implements Runnable {
	private Channel mbox;

	public Consumer(Channel mbox) {
		this.mbox = mbox;
	}

	public void run() {
		Date message;
		while (true) {
				SleepUtilities.nap();
				System.out.println(“Consumer wants to consume.”);
				message = (Date) mbox.receive();
				if (message != null)
						System.out.println(“Consumer consumed” + message);
				}
		}
}
```

Assignment

- 4.2, 4.4, 4.5

End of Chapter 4

Any Question?