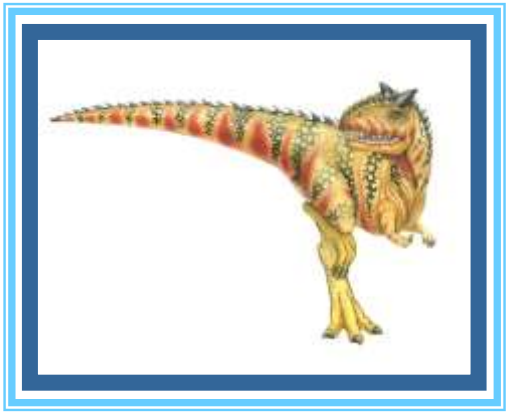
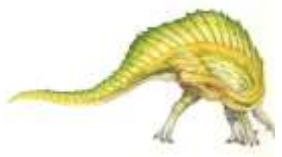
Chapter 4: Threads



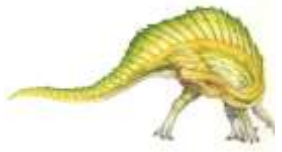
Chapter 4: Threads

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples



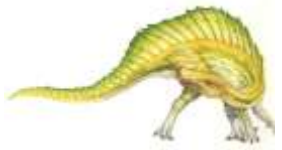
Objectives

- 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, Windows, and Java thread libraries
- To explore several strategies that provide implicit threading
- To examine issues related to multithreaded programming
- To cover operating system support for threads in Windows and Linux

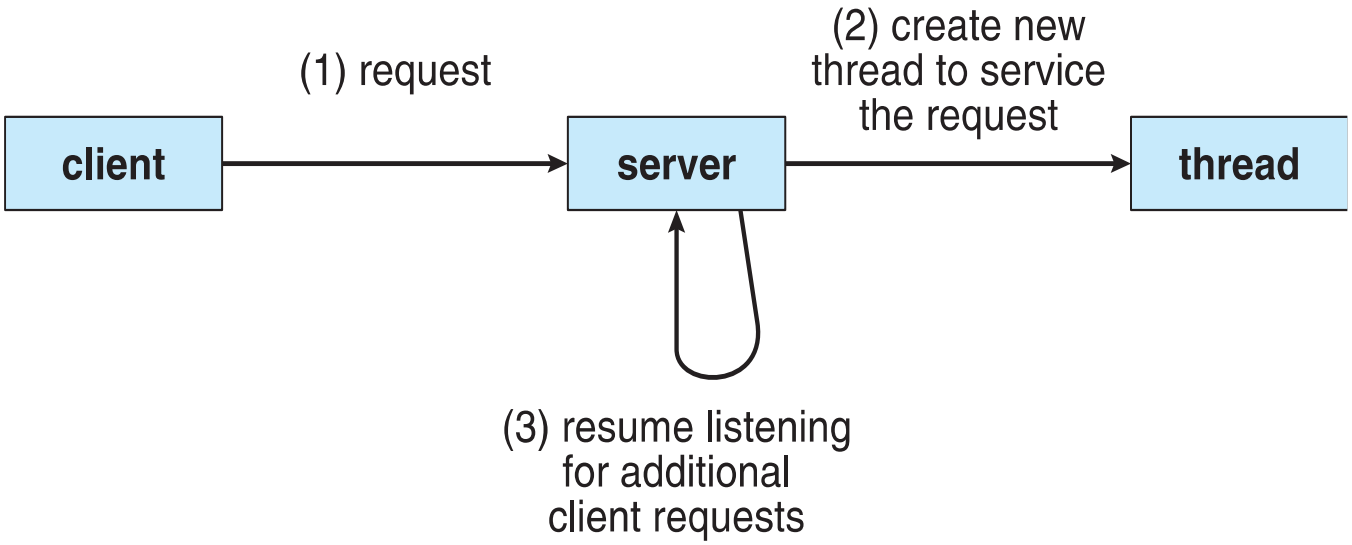


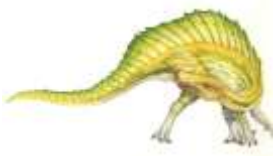
Motivation

- Most modern applications are multithreaded
- 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



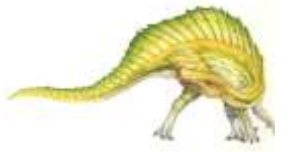
Multithreaded Server Architecture





Benefits

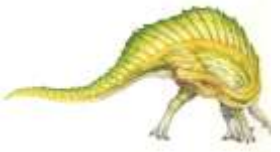
- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multiprocessor architectures



Process vs Threads

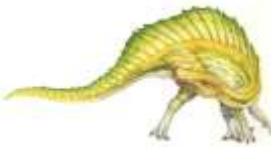
2

- **Similarities**
  - Like processes threads share CPU and only one thread active (running) at a time.
  - Like processes, threads within a processes, threads within a processes execute sequentially.
  - Like processes, thread can create children.
  - And like process, if one thread is blocked, another thread can run.
- **Differences**
  - Unlike processes, threads are not independent of one another.
  - Unlike processes, all threads can access every address in the task .
  - Unlike processes, thread are design to assist one other.
  - As processes may originate from different users they may or may not assist one another.



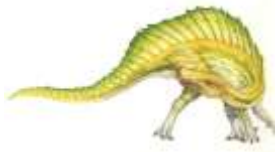
But Why Threads?

- A process with multiple threads make a great server (for example printer server.)
- Because threads can share common data, they do not need to use inter process communication.
- Because of the very nature, threads can take advantage of multiprocessors.
- Threads only need a stack and storage for registers therefore, threads are easy to create.
- Threads use very little resources of an operating system in which they are working.
- Threads do not need new address space, global data, program code or operating system resources.
- Context switching are fast when working with threads. The reason is that we only have to save and/or restore PC, SP and registers.



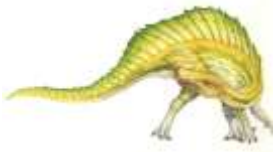
Multicore Programming

- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency



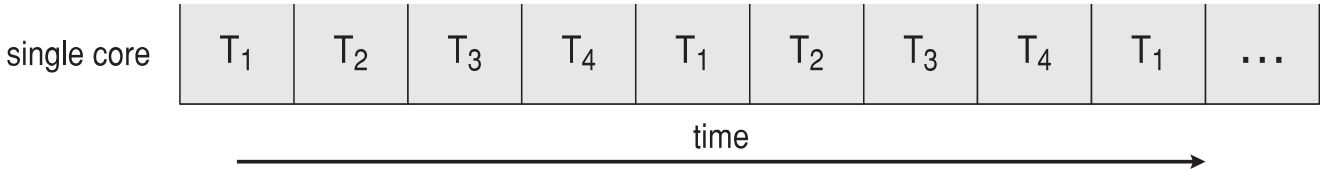
Multicore Programming (Cont.)

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As # of threads grows, so does architectural support for threading
  - CPUs have cores as well as ***hardware threads***
  - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

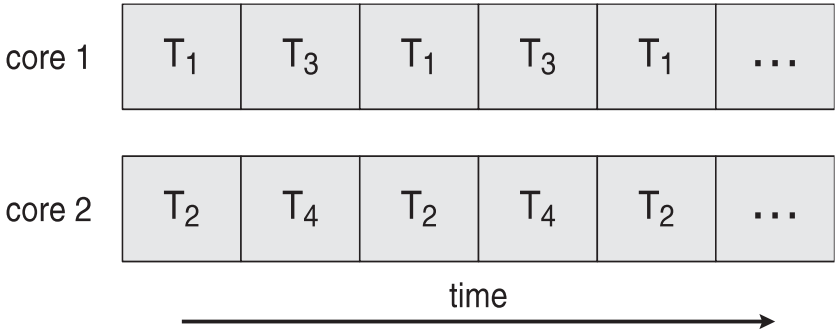


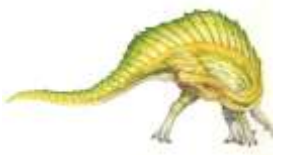
Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**

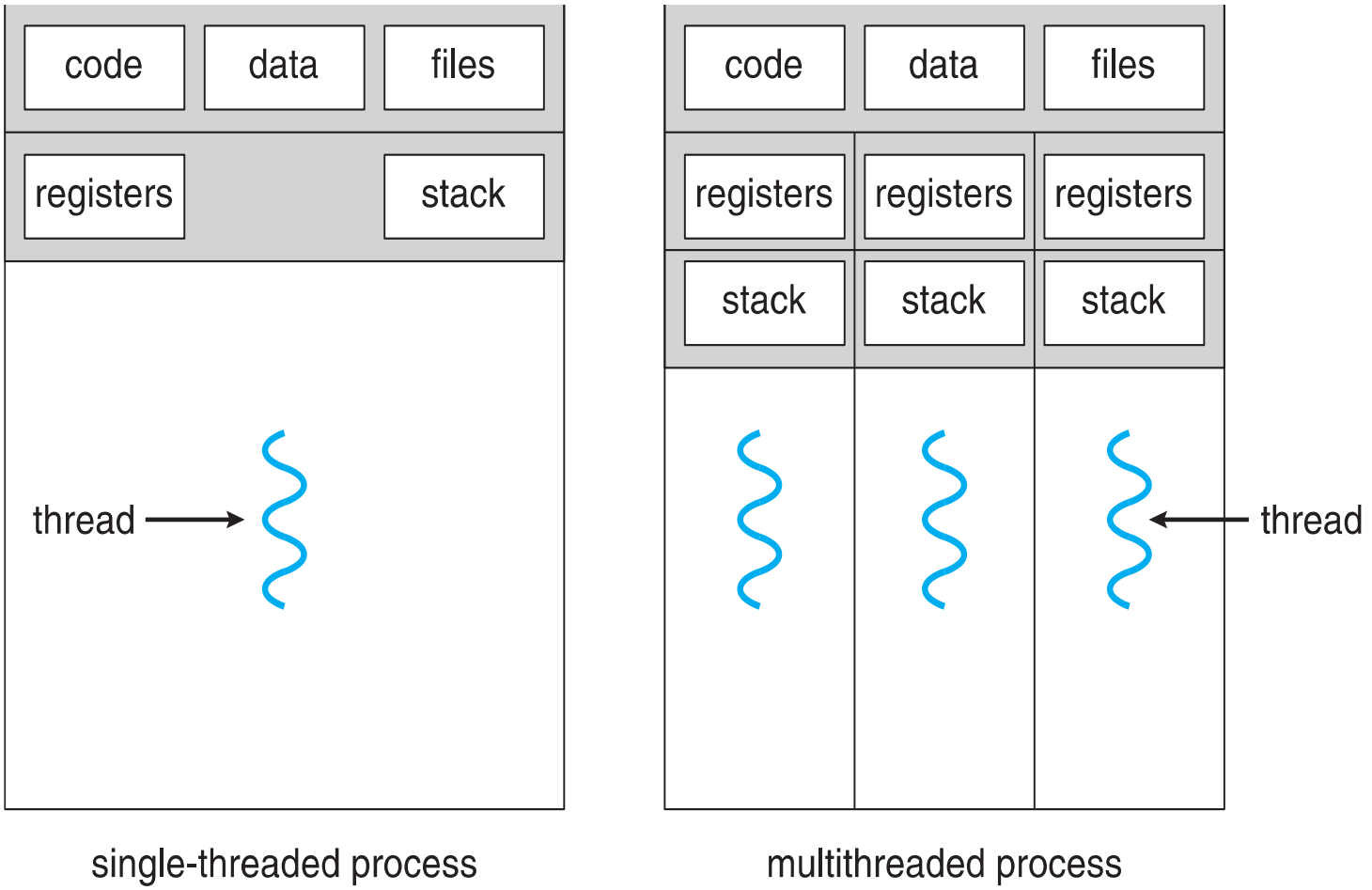


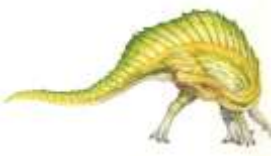
- **Parallelism on a multi-core system:**





Single and Multithreaded Processes





Amdahl's Law

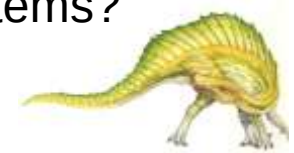
- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- *S* is serial portion
- *N* processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As *N* approaches infinity, speedup approaches 1 / *S*

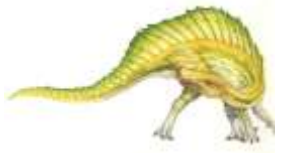
Serial portion of an application has disproportionate effect on performance gained by adding additional cores

- But does the law take into account contemporary multicore systems?



User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

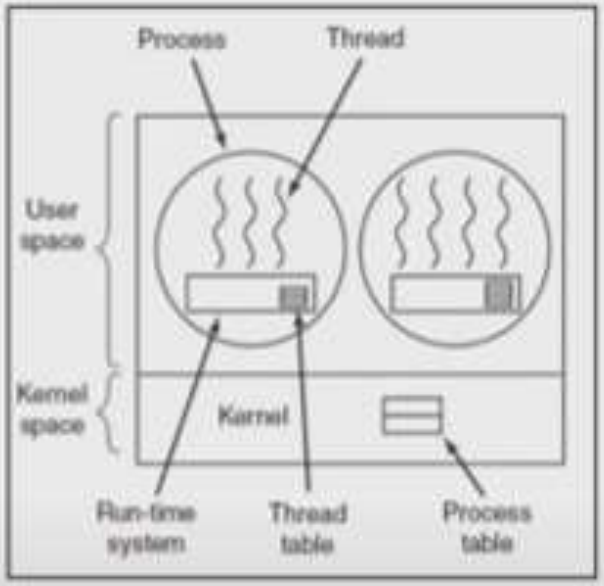


User level threads

- Advantages:
  - Fast (really lightweight)

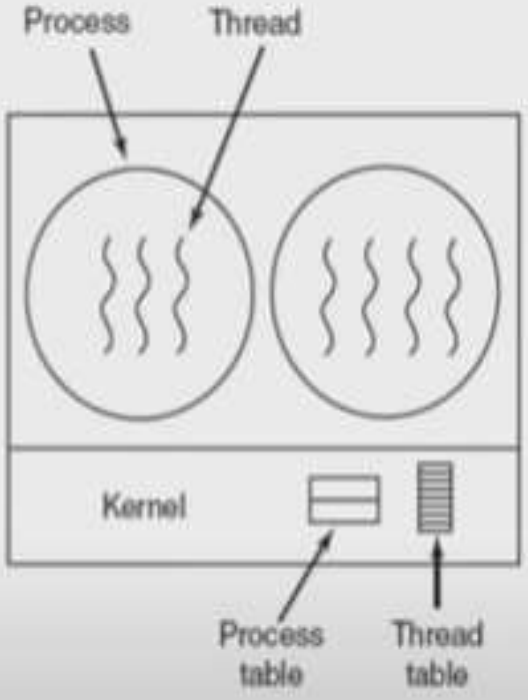
  (no system call to manage threads. The thread library does everything).
  - Can be implemented in an OS that does not support threading.
  - Switching is fast. No switch from user to protected mode.

- Disadvantages:
  - Scheduling can be an issue. (Consider, one thread that is blocked on an IO and another runnable.)
  - Lack of coordination between kernel and threads. (A process with 100 threads competes for a timeslice with a process having just 1 thread.)
  - Requires non-blocking system calls. (If one thread invokes a system call, all threads need to wait)



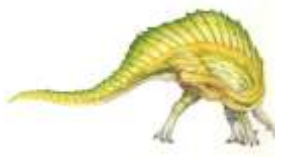
Kernel level threads

- Advantages:
  - Scheduler can decide to give more time to a process having large number of threads than process having small number of threads.
  - Kernel-level threads are especially good for applications that frequently block.
- Disadvantages:
  - The kernel-level threads are slow (they involve kernel invocations.)
  - Overheads in the kernel. (Since kernel must manage and schedule threads as well as processes. It require a full thread control block (TCB) for each thread to maintain information about threads.)



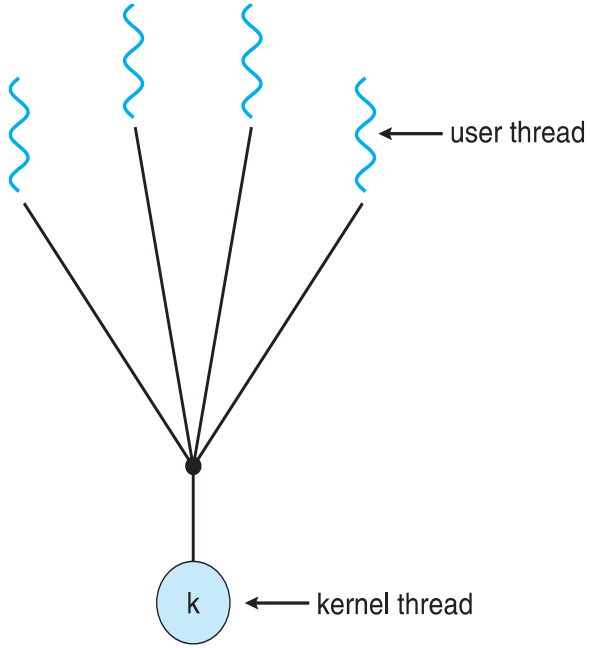
Multithreading Models

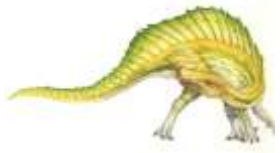
- Many-to-One
- One-to-One
- Many-to-Many



Many-to-One

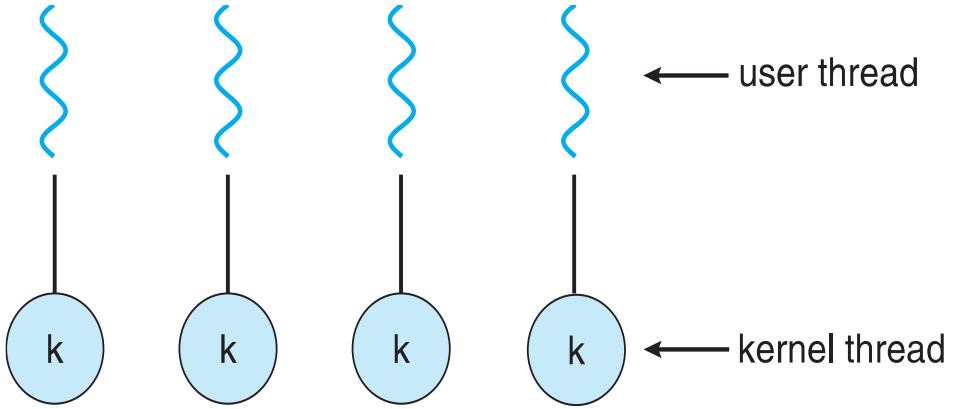
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**

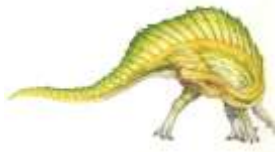




One-to-One

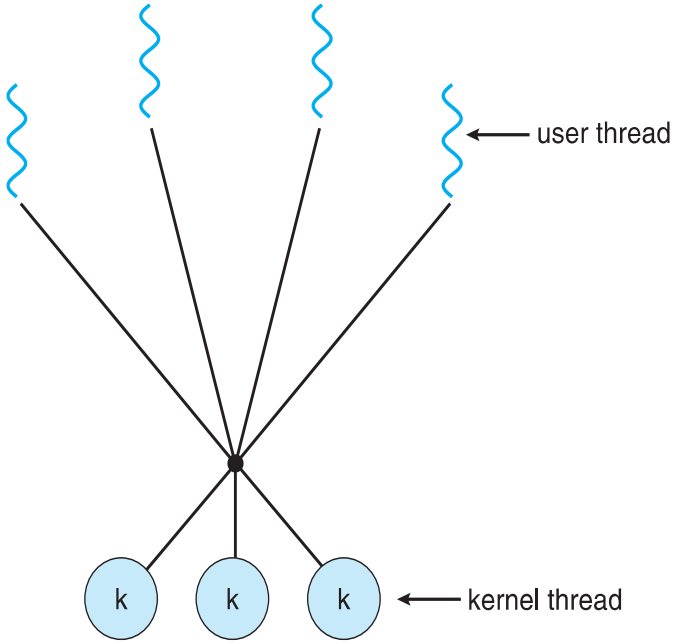
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later

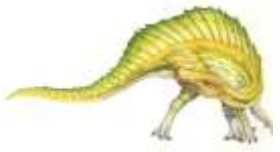




Many-to-Many Model

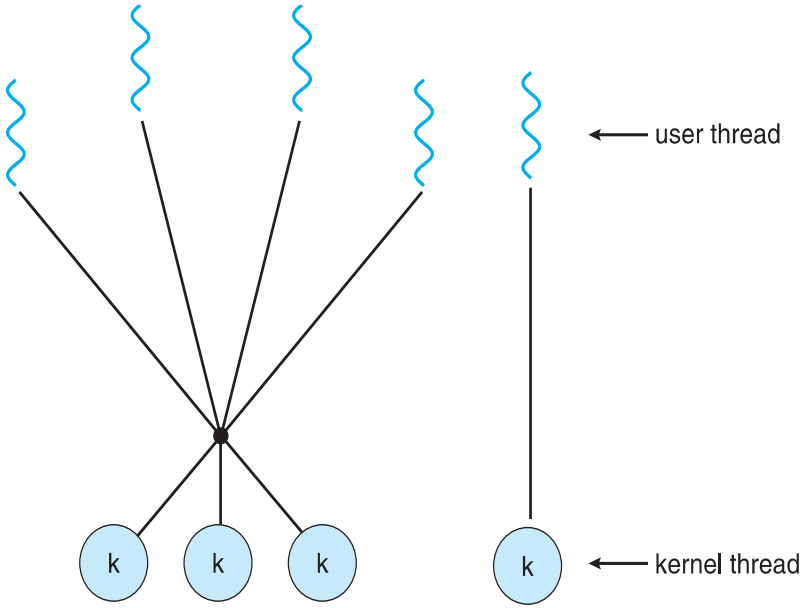
- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package

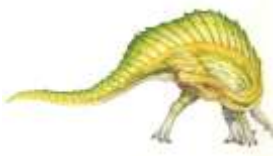




Two-level Model

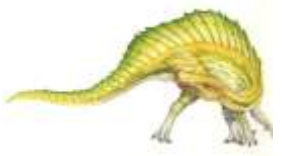
- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier





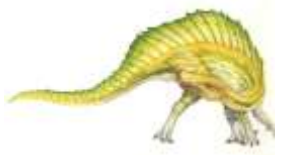
Thread Libraries

- **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
- ***Specification***, not ***implementation***
- 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)

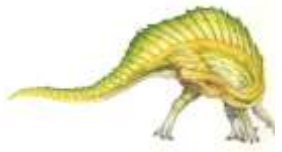


Pthread Example 1

what is the output of the following program?

```
#include<stdio.h>
#include<unistd.h>
#include<pthread.h>
Void myturn()
{
while(1){
Sleep(1);
printf("myturn\n");
}
}
```

```
Void yourturn()
{
while(1){
Sleep(2);
printf("yourturn\n");
}
}
Void main()
{
myturn();
yourturn();

}
```



Pthread Example 2

what is the output of the following program?

```
#include<stdio.h>
#include<unistd.h>
#include<pthread.h>
Void * myturn( void *arg)
{
while(1){
Sleep(1);
printf("myturn\n");
}
return NULL;
}
```

```
Void yourturn()
{
while(1){
Sleep(2);
printf("yourturn\n");
}
}
Void main()
{
pthread_t newthread;
pthread_create(&newthread, NULL,
myturn, NULL);
//myturn();
yourturn();

}
```

Pthread Example 3 pthread _create

what is the output of the following program?

```
#include<stdio.h>
#include<unistd.h>
#include<pthread.h>
Void * myturn( void *arg)
{
for(int  i=0;i<9;i++)
{
Sleep(1);
printf("myturn: %d \n",i);
}
return NULL;
}
```

```
Void yourturn()
{
for(int  i=0;i<5;i++)
){
Sleep(2);
printf("yourturn:%d\n",i);
}
}
Void main()
{
pthread_t newthread;
pthread_create(&newthread, NULL,
myturn, NULL);
//myturn();
yourturn();

}
```

Pthread Example 4 (pthread_ join)

what is the output of the following program?

```
#include<stdio.h>
#include<unistd.h>
#include<pthread.h>
Void * myturn( void *arg)
{
for(int  i=0;i<9;i++)
{
Sleep(1);
printf("myturn: %d \n",i);
}
return NULL;
}
```

```
Void yourturn()
{
for(int  i=0;i<5;i++)
){
Sleep(2);
printf("yourturn:%d\n",i);
}
}
Void main()
{
pthread_t newthread;
pthread_create(&newthread, NULL,
myturn, NULL);
//myturn();
yourturn();
pthread_join(&newthread,NULL);
}
```

Pthread Example 5(passing args to thread function

what is the output of the following program?

```
#include<stdio.h>
#include<unistd.h>
#include<pthread.h>
Void * myturn( void *arg)
{
Int *iptr=(int*) arg;
for(int  i=0;i<9;i++)
{
Sleep(1);
printf("myturn: %d  %d \n",i
,*iptr);
(*iptr)++;
}
return NULL;
}
```

```
Void yourturn()
{
for(int  i=0;i<5;i++)
){
Sleep(2);
printf("yourturn:%d\n",i);
}
}
Void main()
{
pthread_t newthread;
Int  v=5;
pthread_create(&newthread, NULL, myturn,
&v);
//myturn();
yourturn();
pthread_join(&newthread,NULL);
Printf("threads done and final value of
v:%d",v);
}
```

Pthread Example 6(returning values from thread function

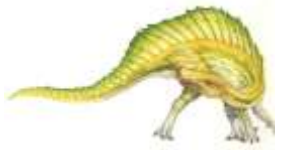
what is the output of the following program?

```
#include<stdio.h>
#include<unistd.h>
#include<pthread.h>
Void * myturn( void *arg)
{
Int *iptr=
(int*)malloc(sizeof(int));
*iptr=3;
for(int  i=0;i<9;i++)
{
Sleep(1);
printf("myturn: %d  %d \n",i
,*iptr);
(*iptr)++;
}
return NULL;
}
```

```
Void yourturn()
{
for(int  i=0;i<5;i++)
){
Sleep(2);
printf("yourturn:%d\n",i);
}
}
Void main()
{
pthread_t newthread;
Int  v=5;
pthread_create(&newthread, NULL, myturn, &v);
//myturn();
yourturn();
pthread_join(&newthread,NULL);
Printf("threads done and final value of v:%d",v);
}
```

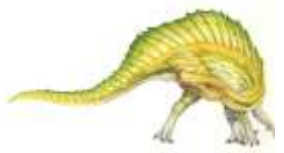
Pthreads Example

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

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;
   }
```



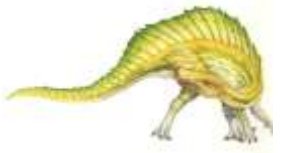
Pthreads Example (Cont.)

```
   /* 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);
}
```



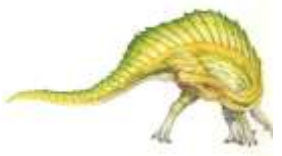
Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```



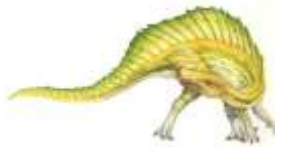
Windows Multithreaded C Program

```
#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;

   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;
   }
```



Windows 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);
   }
}
```

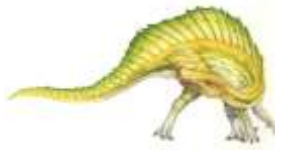


Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:

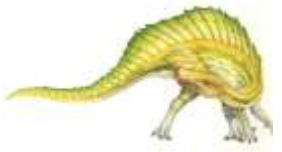
```
public interface Runnable
{
    public abstract void run();
}
```

  - 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);
   }
}
```

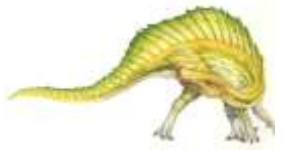


Java Multithreaded Program (Cont.)

```
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 {
         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>"); }
}
```

Implicit Threading

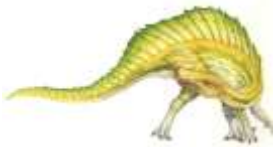
- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Three methods explored
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch
- Other methods include Microsoft Threading Building Blocks (TBB), `java.util.concurrent` package



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
  - Separating task to be performed from mechanics of creating task allows different strategies for running task
    - i.e.Tasks could be scheduled to run periodically
- Windows API supports thread pools:

```
DWORD WINAPI PoolFunction(AVOID Param) {
    /*
    * this function runs as a separate thread.
    */
}
```



End of Chapter 4