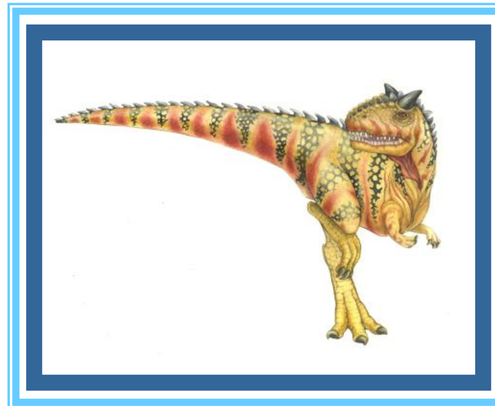
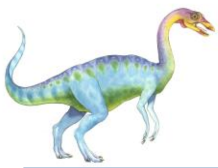
Chapter 4: Threads & Concurrency

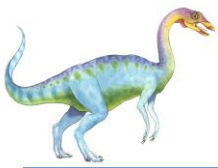


Silberschatz, Galvin and Gagne ©2018



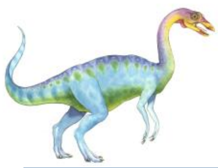
Chapter 4: Threads

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues



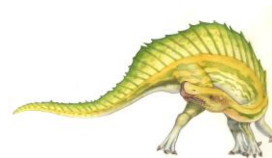
Objectives

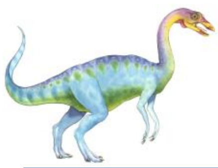
- Identify the basic components of a thread, and contrast threads and processes
- Describe the benefits and challenges of designng multithreaded applications
- Design multithreaded applications using the Pthreads, Java, and Windows threading APIs



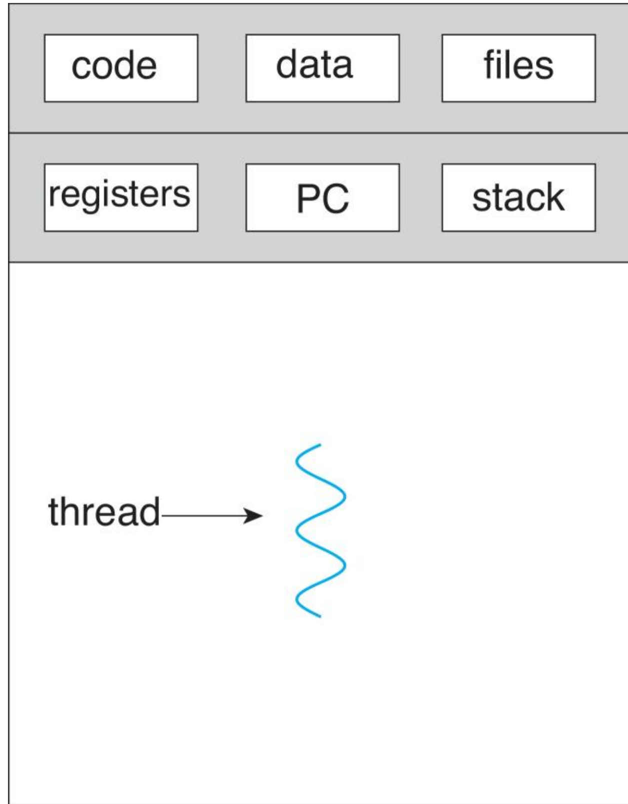
Motivation

- 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

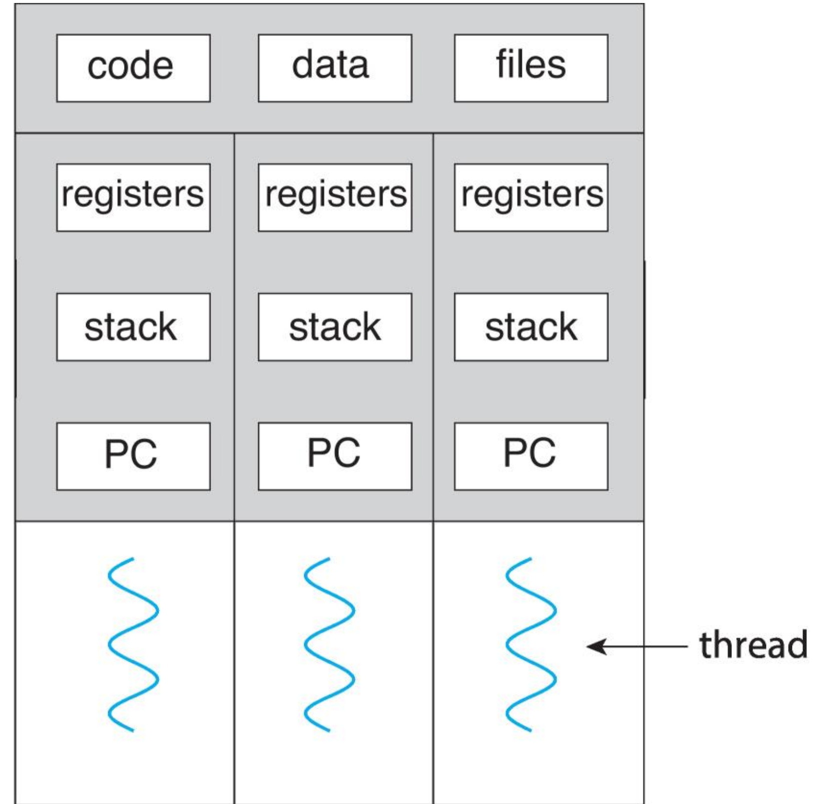




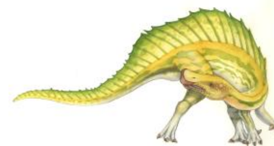
Single and Multithreaded Processes

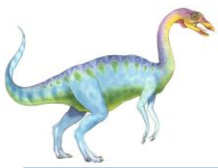


single-threaded process



multithreaded process





Multithreaded Server Architecture

(1) request

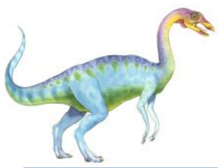
(2) create new thread to service the request

client

server

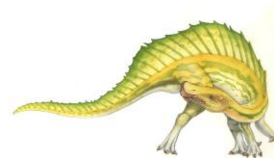
thread

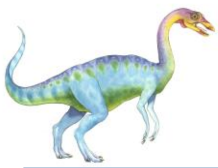
(3) resume listening for additional client requests



Benefits

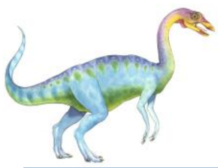
- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower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





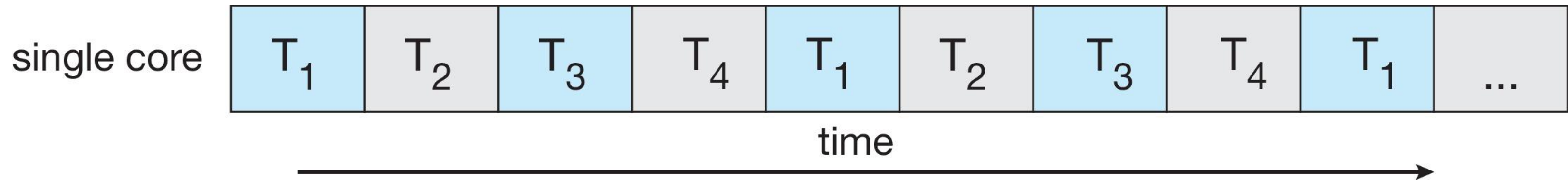
Multicore Programming

- **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

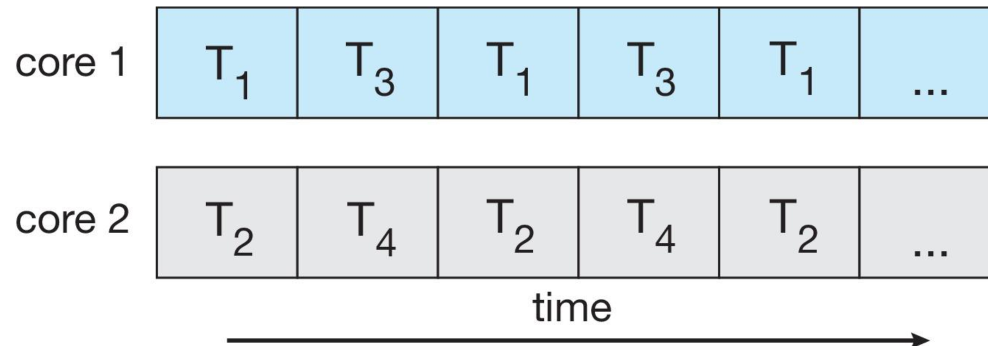


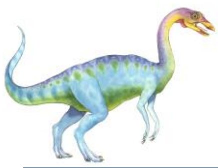
Concurrency vs. Parallelism

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



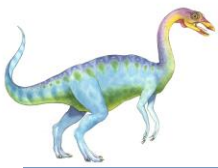
- **Parallelism on a multi-core system:**



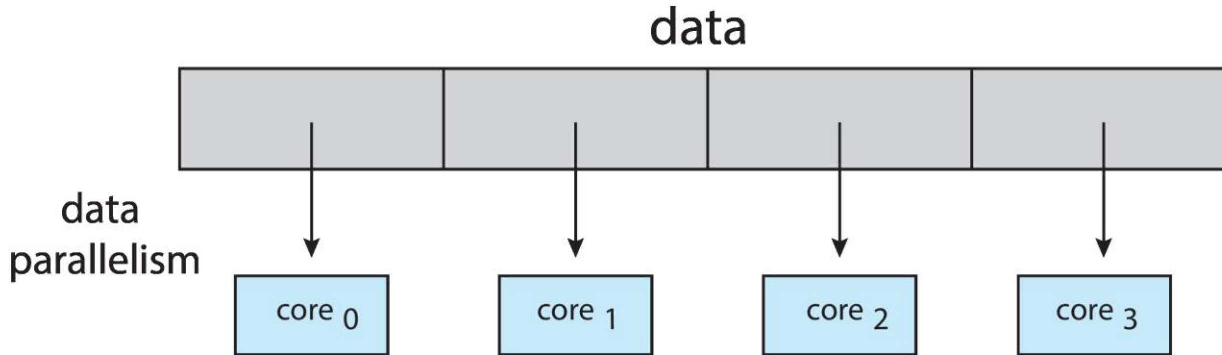


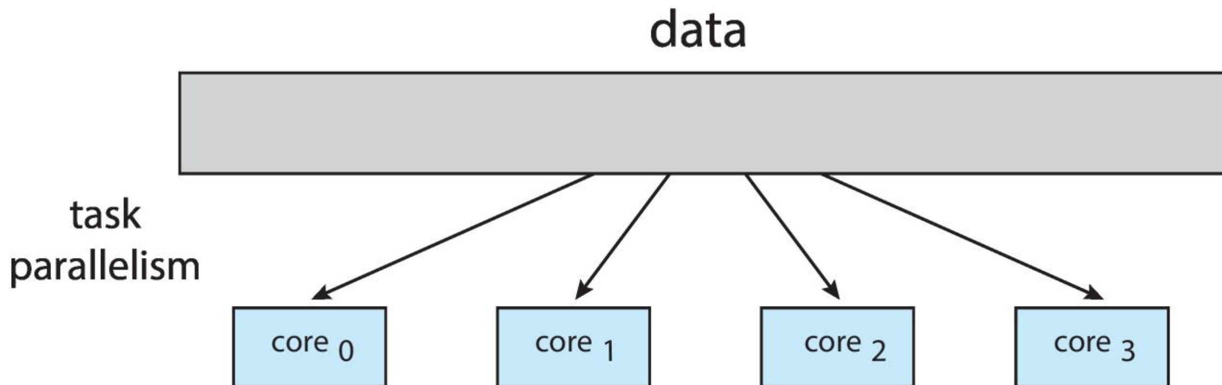
Multicore Programming

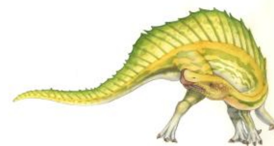
- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation



Data and Task Parallelism







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

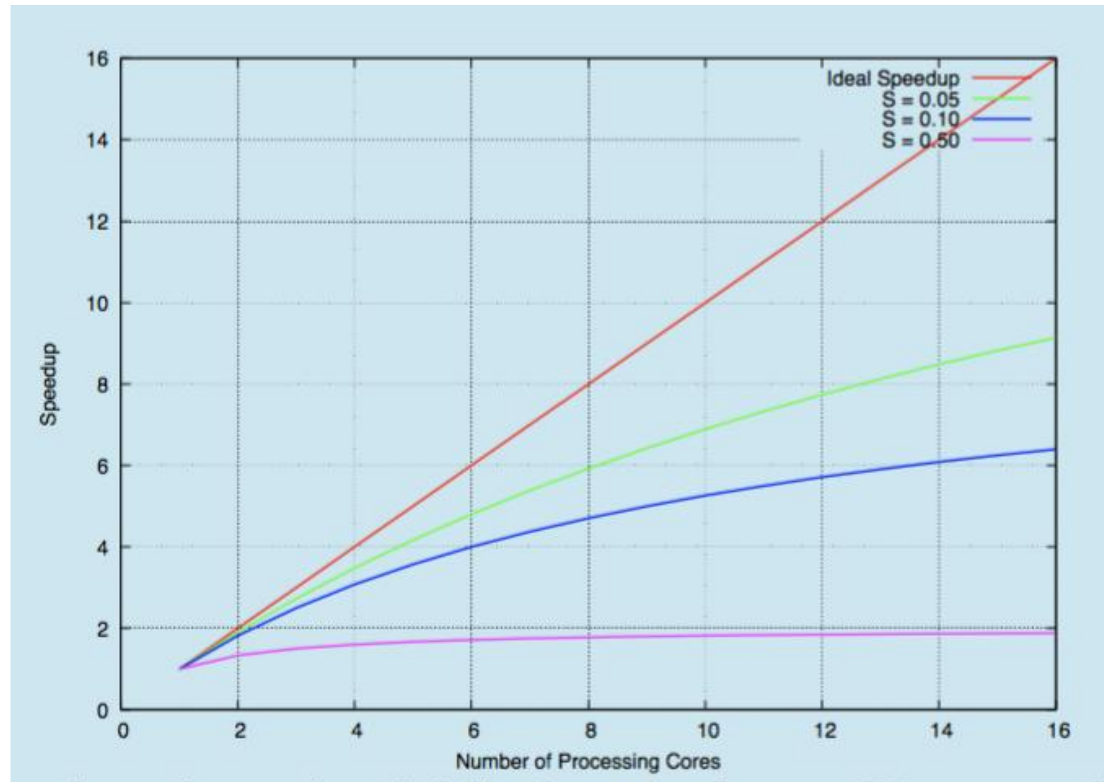
$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

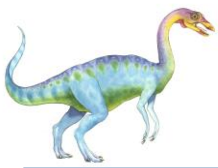
- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As *N* approaches infinity, speedup approaches 1 / *S*

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?

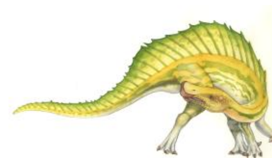
Amdahl's Law

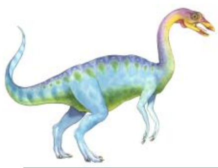




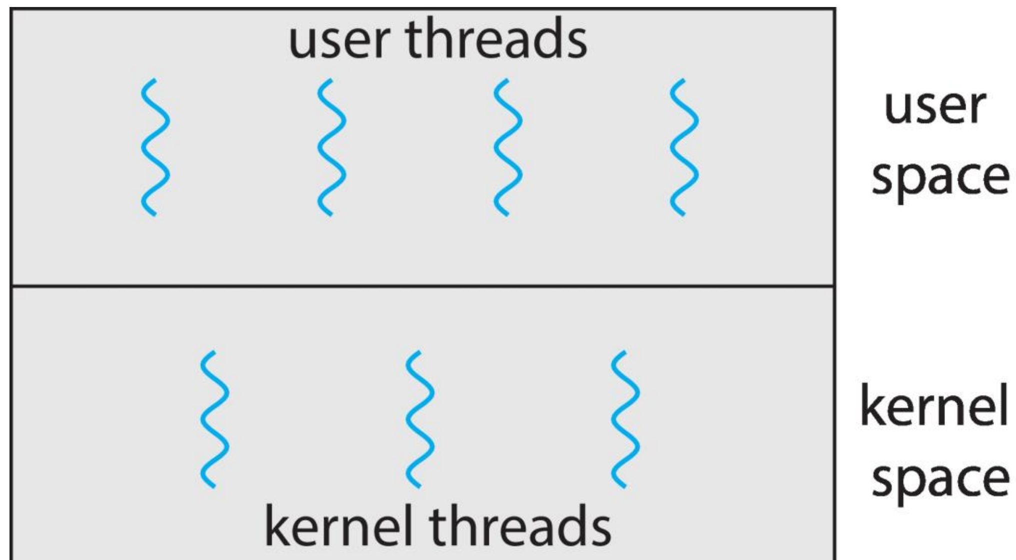
User Threads and Kernel Threads

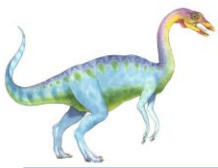
- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Linux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





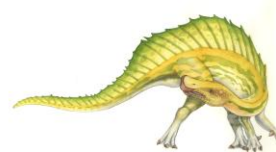
User and Kernel Threads





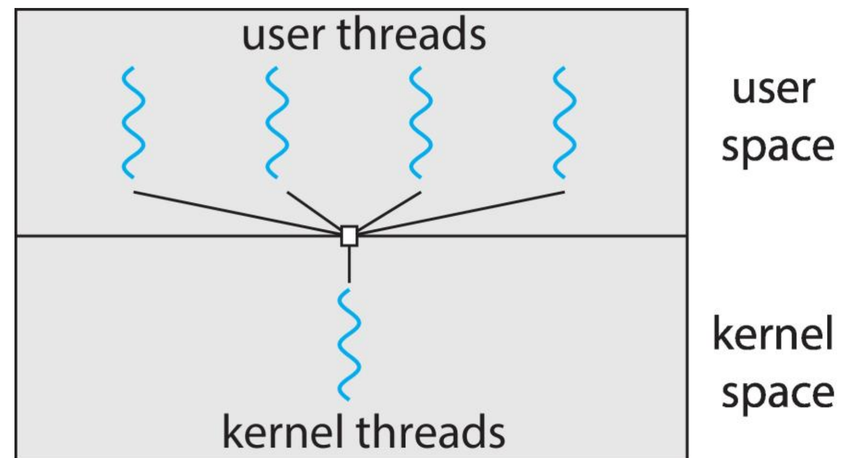
Multithreading Models

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



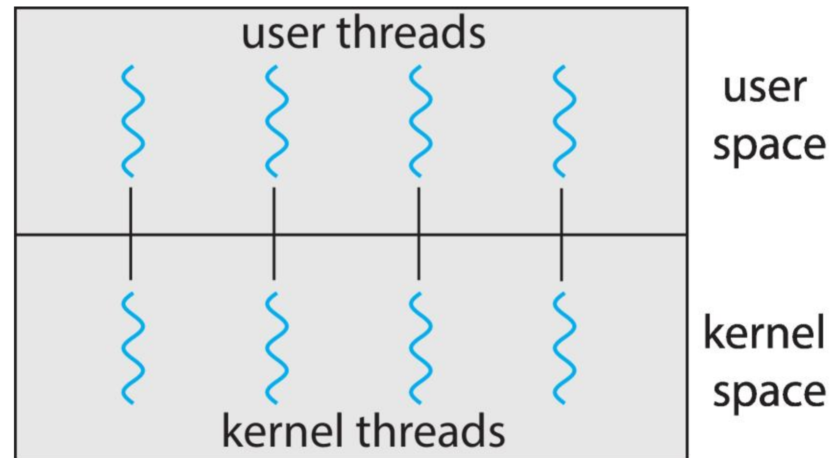
Many-to-One

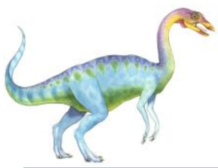
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable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



One-to-One

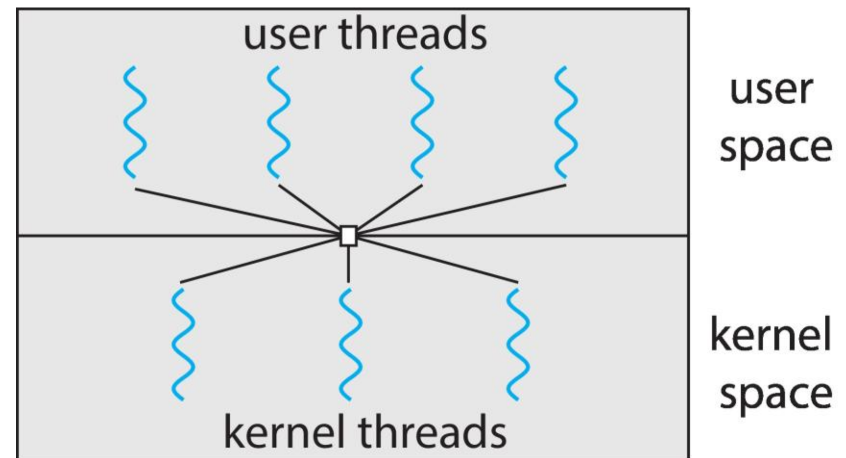
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux





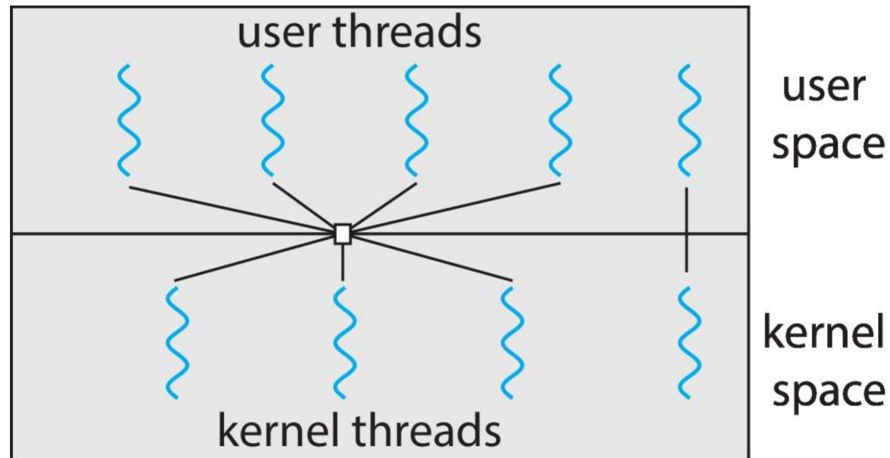
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Windows with the *ThreadFiber* package
- Otherwise not very common



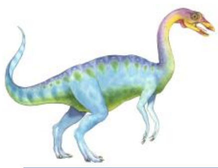
Two-level Model

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



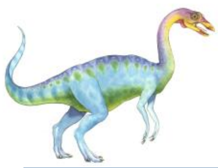
Thread Libraries

- **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 (Linux & Mac OS X)

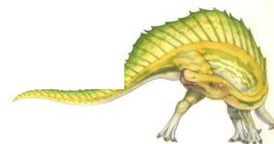


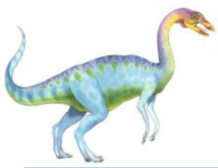
Example

- As an illustrative example, we design a multi threaded program that performs the summation of a non-negative integer in a separate thread
- For example, if N were 5, this function would represent the summation of integers from 0 to 5, which is 15.
- When this program begins, a single thread of control begins in main().
- After some initialization, **main() creates a second thread** that begins control in the **runner() function**.
- **Both threads share the global data sum**.

Pthreads Example

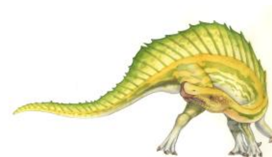
```
#include <pthread.h>
#include <stdio.h>

#include <stdlib.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 */

   /* set the default attributes of the thread */
   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);
}
```

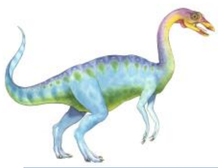




Pthreads Example (cont)

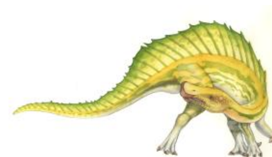
```
/* The thread will execute in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

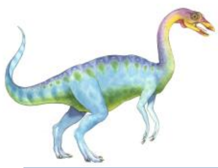




Pthreads Code for Joining 10 Threads

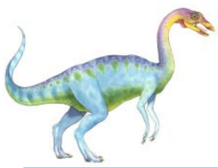
```
#define NUM_THREADS 10

/* 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 will execute in this function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 1; i <= Upper; i++)
      Sum += i;
   return 0;
}
```



Windows Multithreaded C Program (Cont.)

```
int main(int argc, char *argv[])
{
  DWORD ThreadId;
  HANDLE ThreadHandle;
  int Param;

  Param = atoi(argv[1]);
  /* 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 */

   /* 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
- Because **Java has no notion of global data**, access to **shared data must be explicitly arranged** between threads.
- Java threads may be created by:
  - Extending Thread class or
  - Implementing the Runnable interface

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

  - Standard practice is to implement Runnable interface

Java Threads

Implementing Runnable interface:

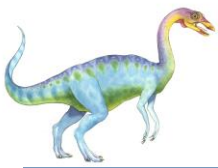
```
class Task implements Runnable
{
   public void run() {
      System.out.println("I am a thread.");
   }
}
```

Creating a thread:

```
Thread worker = new Thread(new Task());
worker.start();
```

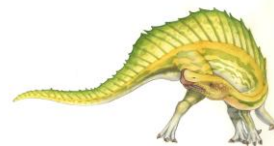
Waiting on a thread:

```
try {
   worker.join();
}
catch (InterruptedException ie) { }
```



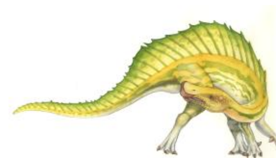
Creating Threads in Java

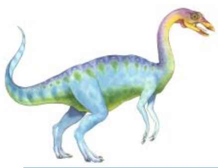
- There are two techniques for creating threads in a Java program.
  - One approach is to create a new class that is derived from the Thread class and to override its run() method.
  - An alternative—and more commonly used— technique is to define a class that implements the Runnable interface.
    - The code implementing the run() method is what runs as a separate thread.
    - Thread creation is performed by creating an object instance of the Thread class and passing the constructor a Runnable object.
    - The start() method creates the new thread
    - start() allocates memory and initializes a new thread in the JVM.
    - It also calls the run() method, making the thread eligible to be run by the JVM.
    - The join() method in Java is equivalent to pthread join() and WaitForSingleObject()



Sharing Data Between Threads in Java

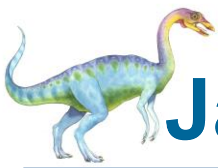
- If two or more threads are to share data in a Java program, the sharing occurs by passing references to the shared object to the appropriate threads.
- In our example the main thread and the summation thread share the object instance of the Sum class.
- This shared object is referenced through the appropriate getSum() and setSum() methods





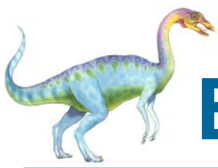
Java Multithreaded Program

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



Extra Example on Threads in Java

(1) Inheriting from the `Thread` class

- The general approach is
  - (1) **Define a class** by **extending** the `Thread` class and **overriding** the `run` method.
    - In the run method, you should write the code that you wish to run when this particular thread has started.
  - (2) **Create an instance** of the above class
  - (3) **Start running the instance** using the `start` method that is defined in `Thread`.

Example

```
public class WhereAmI extends Thread {
    int n;
// constructor
    public WhereAmI(int number) {
        n = number;
    }
// override the run method
    public void run() {
        for (int i = 0; i < 100; i++)
            System.out.println("I'm in thread " + n);

    }
}
```

```
public class ThreadTester {
    public static void main(String[] args) {
        // create the threads
        WhereAmI place1 = new WhereAmI(1);
        WhereAmI place2 = new WhereAmI(2);
        WhereAmI place3 = new WhereAmI(3);
        // start the threads
        place1.start();
        place2.start();
        place3.start();
    }
}
```

The Output

```
I'm in thread 3
I'm in thread 1
I'm in thread 3
I'm in thread 1
I'm in thread 3
I'm in thread 1
I'm in thread 3
I'm in thread 2
I'm in thread 3
I'm in thread 2
I'm in thread 3
I'm in thread 2
```

Extra Example on Threads in Java (Cont.)

(2) Implementing the `Runnable` interface

- The general approach is
  - (1) **Define a class** that **implements** `Runnable` and **overriding** the `run` method.
  - (2) **Create an instance** of the above class.
  - (3) **Create a thread** that runs this instance.
  - (4) **Start running the instance** using the `start` method.

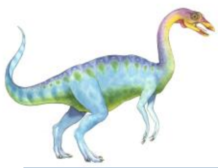
Example

```
public class WhereAmI2 implements Runnable{
    int n;
    // constructor
    public WhereAmI2(int number) {
        n = number;
    }
    // override the run method
    public void run() {
        for (int i = 0; i < 100; i++)
            System.out.println("I'm in thread " + n);

    }
}
```

```
public class ThreadTester2 {
    public static void main(String[] args) {
        // create a runnable objects,
        // and the thread to run them.
        WhereAmI2 place1 = new WhereAmI2(1);
        Thread thread1 = new Thread(place1);
        WhereAmI2 place2 = new WhereAmI2(2);
        Thread thread2 = new Thread(place2);
        WhereAmI2 place3 = new WhereAmI2(3);
        Thread thread3 = new Thread(place3);
        // start the threads
        thread1.start();
        thread2.start();
        thread3.start();
    }
}
```

The output

```
I'm in thread 3
I'm in thread 1
I'm in thread 3
I'm in thread 1
I'm in thread 3
I'm in thread 1
I'm in thread 3
I'm in thread 2
I'm in thread 3
I'm in thread 2
I'm in thread 3
I'm in thread 2
```



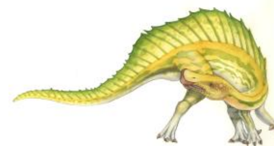
OpenMP

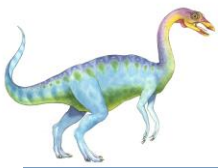
- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel

`#pragma omp parallel`

Create as many threads as there are cores

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

int main(int argc, char *argv[])
{
  /* sequential code */

  #pragma omp parallel
  {
    printf("I am a parallel region.");
  }

  /* sequential code */

  return 0;
}
```

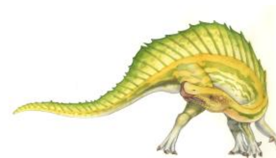




OpenMP (Continoue)

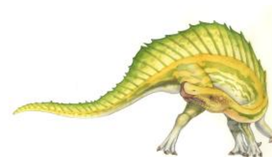
- Run the for loop in parallel

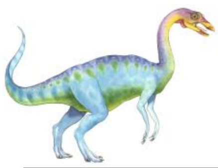
```
#pragma omp parallel for
for (i = 0; i < N; i++) {
   c[i] = a[i] + b[i];
}
```



Threading Issues

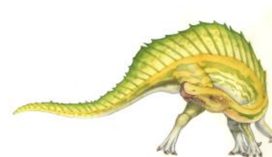
- Semantics of **fork()** and **exec()** system calls
- Signal handling
  - Synchronous and asynchronous
- Thread cancellation of target thread
  - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations





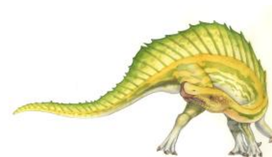
Semantics of fork() and exec()

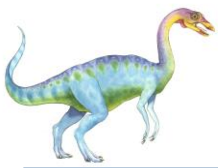
- Does **`fork()`** duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork
- **`exec()`** usually works as normal – replace the running process including all threads
  - That is, if a thread invokes the exec() system call, the program specified in the parameter to exec() will replace the entire process—including all threads.



Signal Handling

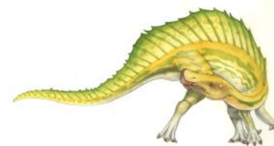
- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled by one of two signal handlers:
     1. default
     2. user-defined
- Every signal has **default handler** that kernel runs when handling signal
  - **User-defined signal handler** can override default
  - For single-threaded, signal delivered to process

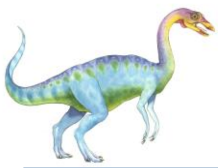




Synchronous Signals

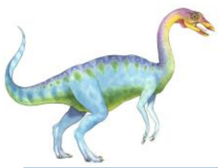
- A signal may be received either synchronously or asynchronously
- Examples of synchronous signal include
  - illegal memory access and
  - division by 0.
- If a running program performs either of these actions, a signal is generated.
- Synchronous signals are delivered to the same process that performed the operation that caused the signal (that is the reason they are considered synchronous).





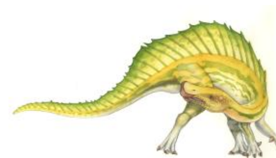
Asynchronous Signals

- When a signal is generated by an event external to a running process, that process receives the signal asynchronously.
- Examples of such signals include terminating a process with specific keystrokes (such as **<control><C>)** and having a timer expire.
- Typically, an asynchronous signal is sent to another process.



Signal Handling (Cont.)

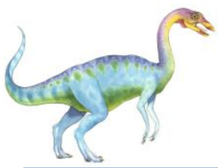
- Where should a signal be delivered for multi-threaded?
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process



Thread Cancellation

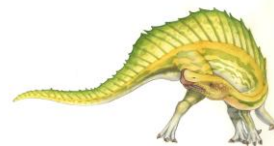
- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Pthread code to create and cancel a thread:

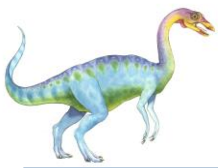
```
pthread_t tid;

/* 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 Cancellation (Cont.)

- Terminating a thread before it has finished
- For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled.
- Another situation might occur when a user presses a button on a web browser that stops a web page from loading any further.
  - Often, a web page loads using several threads—each image is loaded in a separate thread.
  - When a user presses the stop button on the browser, all threads loading the page are canceled.



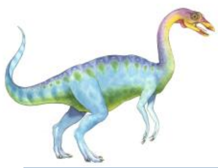


Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches **cancellation point**
    - I.e. `pthread_testcancel()`
    - Then **cleanup handler** is invoked
- On Linux systems, thread cancellation is handled through signals



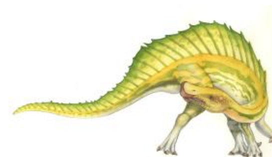
Thread Cancellation in Java

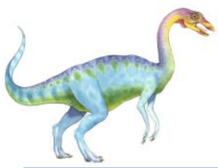
- Deferred cancellation uses the **`interrupt()`** method, which sets the interrupted status of a thread.

```
Thread worker;

 . . .

/* set the interruption status of the thread */
worker.interrupt()
```

- A thread can then check to see if it has been interrupted:

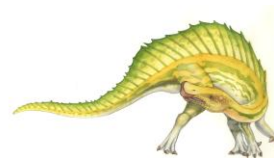
```
while (!Thread.currentThread().isInterrupted()) {
    . . .
}
```

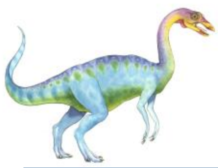




Thread-Local Storage

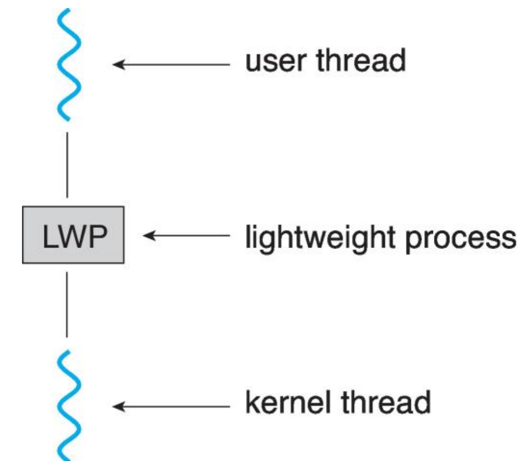
- **Thread-local storage** (**TLS**) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to `static` data
  - TLS is unique to each thread





Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – **lightweight process** (**LWP**)
  - Appears to be a virtual processor on which process can schedule user thread to run
  - Each LWP attached to kernel thread
  - How many LWPs to create?
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library
- This communication allows an application to maintain the correct number kernel threads



End of Chapter 4