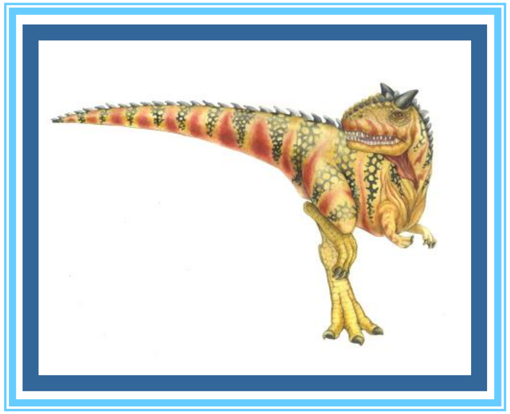
Chapter 4: Threads



Chapter 4: Threads

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

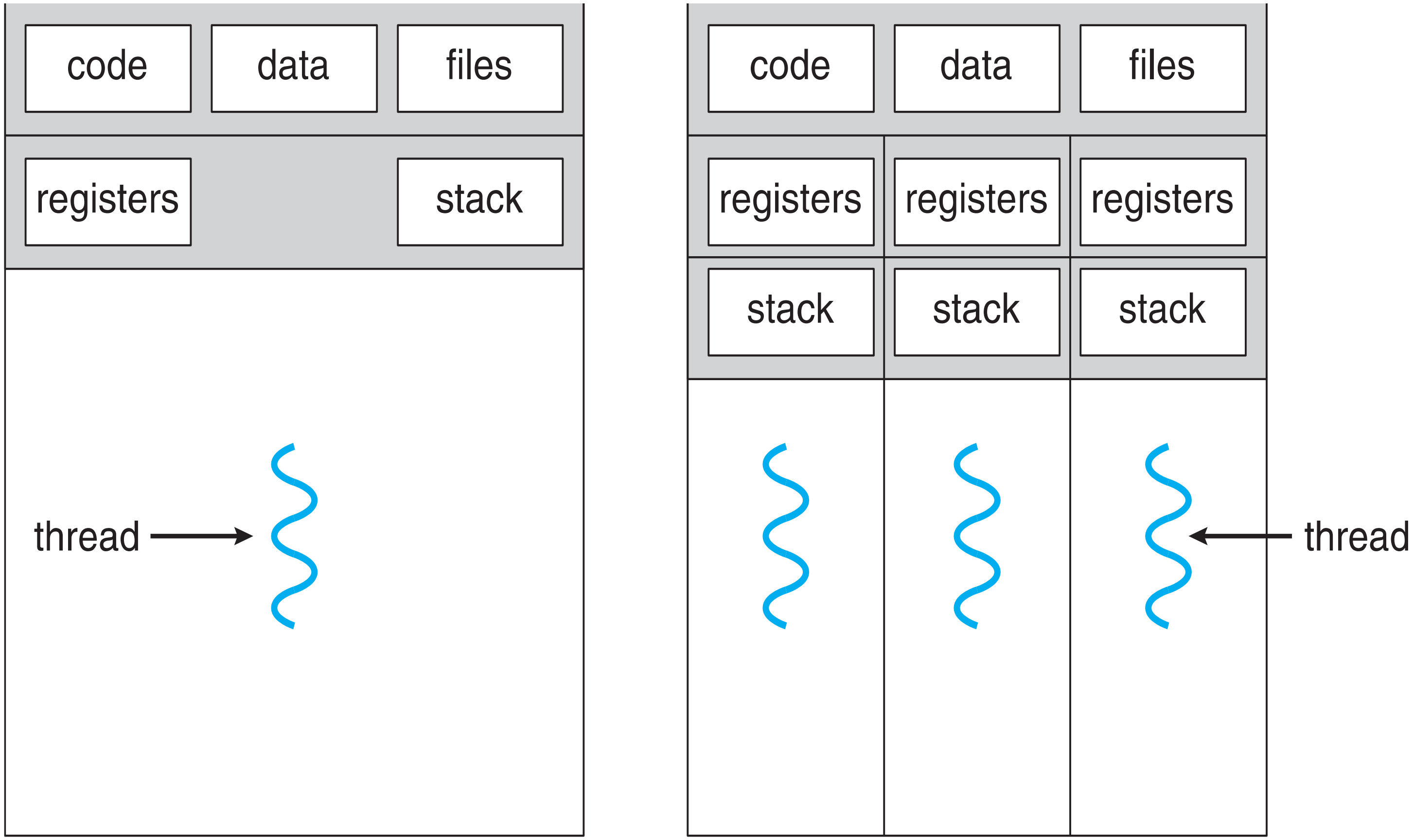
Motivation

- Most modern applications are multithreaded
- Threads run within application
- ***Process creation*** **is** **heavy-weight** **while** ***thread creation*** **is** **light-weight**
- Can simplify code, increase efficiency
- **Kernels are generally multithreaded**

What is?

- A thread is a basic unit of CPU utilization; **a fundamental unit of CPU utilization** that forms the basis of multithreaded computer systems
- It comprises
  - a thread ID,
  - a program counter,
  - a register set,
  - a stack.
- It shares with other threads belonging to the same process
  - its **code section**,
  - **data section**,
  - and other operating-system resources, such as **open files** and **signals.**

Single and Multithreaded Processes



single-threaded process

multithreaded process

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*
  - **SM** and **MP** techniques must be explicitly arranged by the programmer
  - Threads **share** the **memory** and the **resources** of the process **by default**
- **Economy –** cheaper than process creation, **thread switching** lower overhead than **context switching**
  - In Solaris, creating a process is about **thirty times slower** than is creating a thread, and context switching is about **five times slower**.
- **Scalability –** process can take advantage of multiprocessor architectures

4.2 Multicore Programming

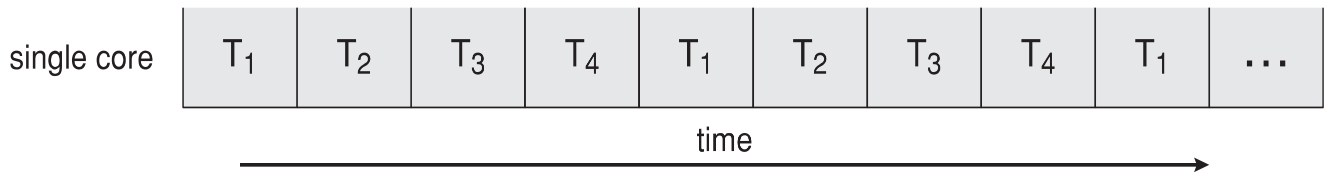
- Each core appears as a separate processor to the operating system
- **Multicore** or **multiprocessor** systems putting pressure on programmers, **challenges** include:
  - **Identifying tasks**
    - This involves examining applications to find areas that can be divided into separate, concurrent tasks
  - **Data splitting**
    - Data accessed and manipulated by the tasks must be divided to run on separate cores
  - **Balance**
    - Tasks should perform **equal work of equal value**.
  - **Data dependency**
    - The data accessed by the tasks must be examined for dependencies between two or more tasks. Execution of the tasks is **synchronized** to accommodate the data dependency.

4.2 Multicore Programming

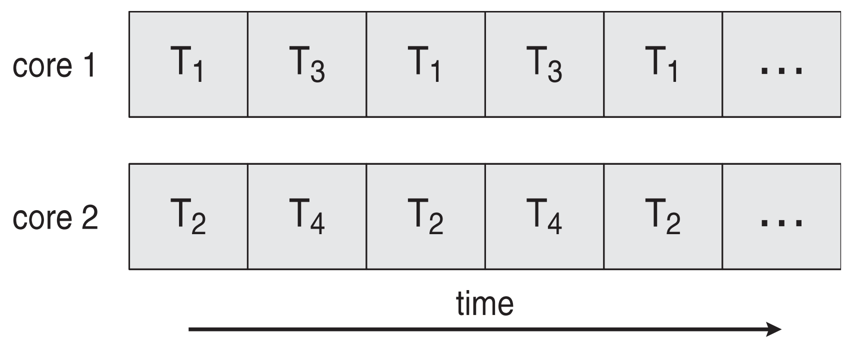
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - The execution of the threads will be **interleaved over time**
  - Single processor / core, scheduler providing concurrency
  - In non-SMP systems (Single core systems) CPU schedulers were designed to provide the illusion of parallelism by rapidly switching between processes

Concurrency vs. Parallelism

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

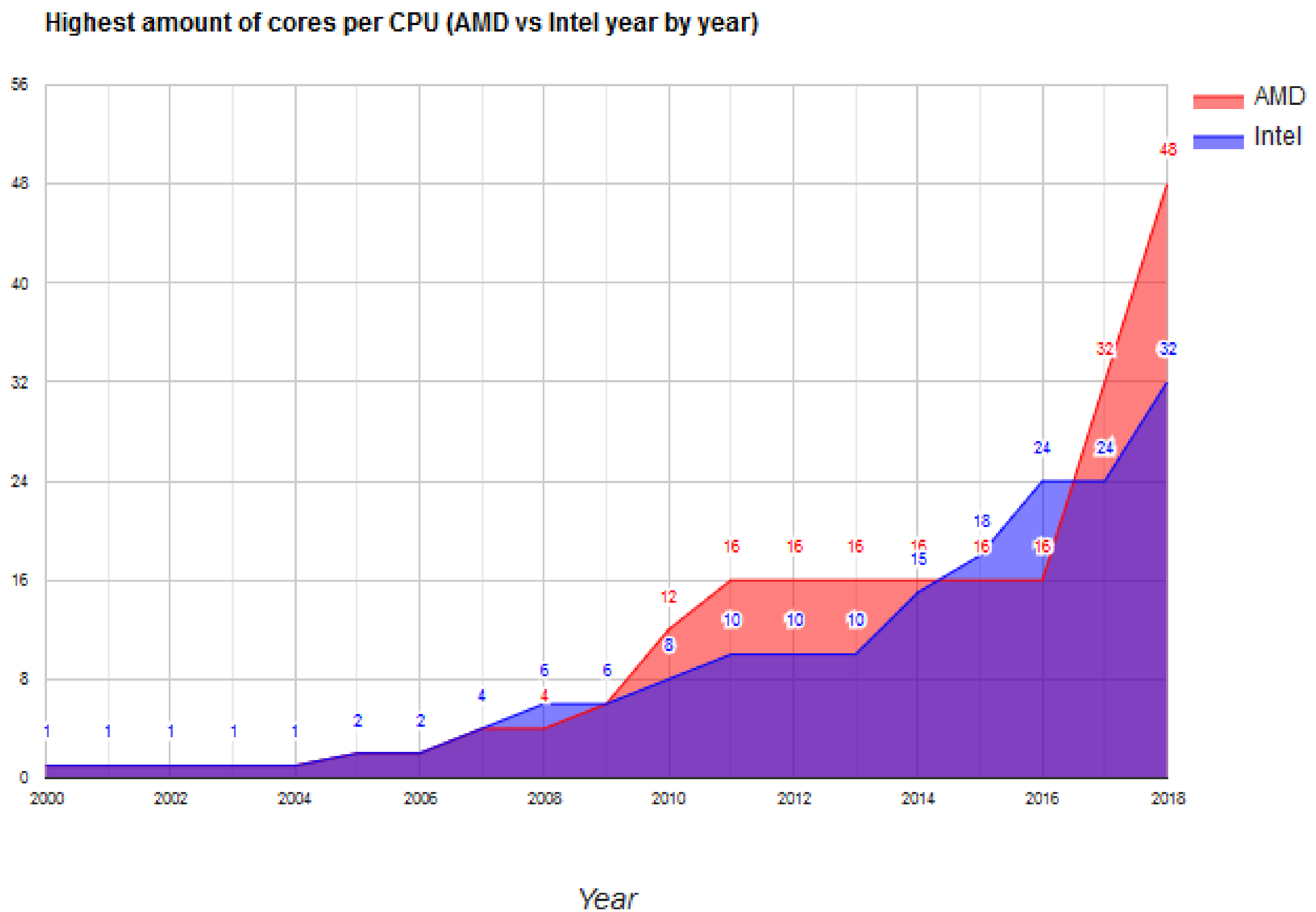


- **Parallelism on a multi-core system:**



Multicore Programming (Cont.)

- As # of threads grows, so does architectural support for threading



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

Multicore Programming (Cont.)

- **Types of parallelism**
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
    - For example, summing the contents of an array of size *N*. On a single-core system, one thread would simply sum the elements [0] . . . [*N* − 1].
    - On a dual-core system, however, thread *A*, running on core 0, could sum the elements [0] . . . [*N*/2 − 1] while thread *B*, running on core 1, could sum the elements [*N*/2] . . . [*N* − 1].
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
    - For example, two threads, each performing a unique statistical operation on the array of elements. The threads again are operating in parallel on separate computing cores, but each is performing a **unique operation**.

4.3 Multithreading Models
User Threads and Kernel Threads

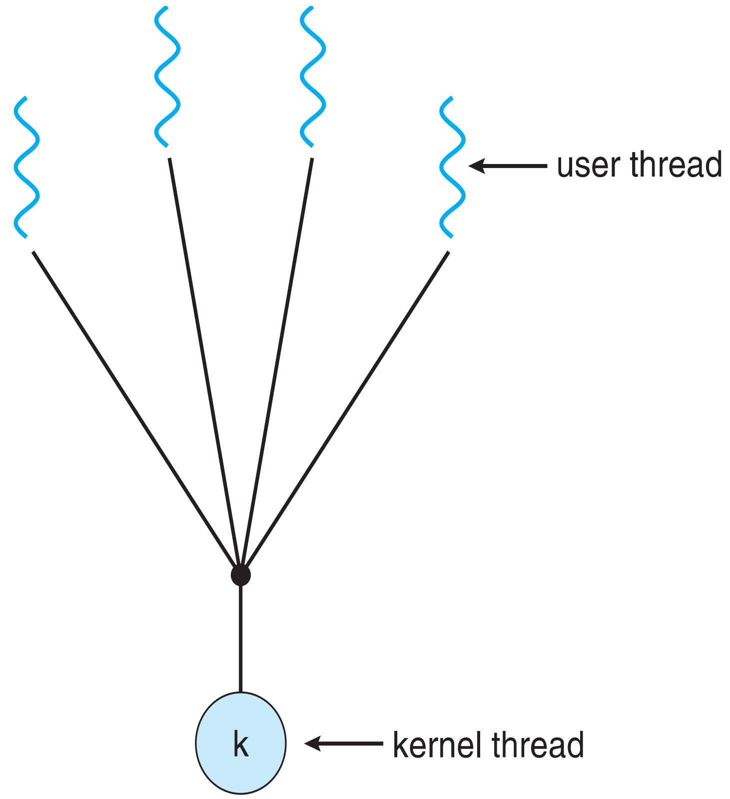
- **User threads** - management done by user-level threads library
  - Three primary thread libraries:
    - POSIX **Pthreads**
    - Windows threads
    - Java threads
- **Kernel threads** - Supported by the Kernel and managed by the OS
  - Examples – virtually all general purpose operating systems,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

4.3 Multithreading Models

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

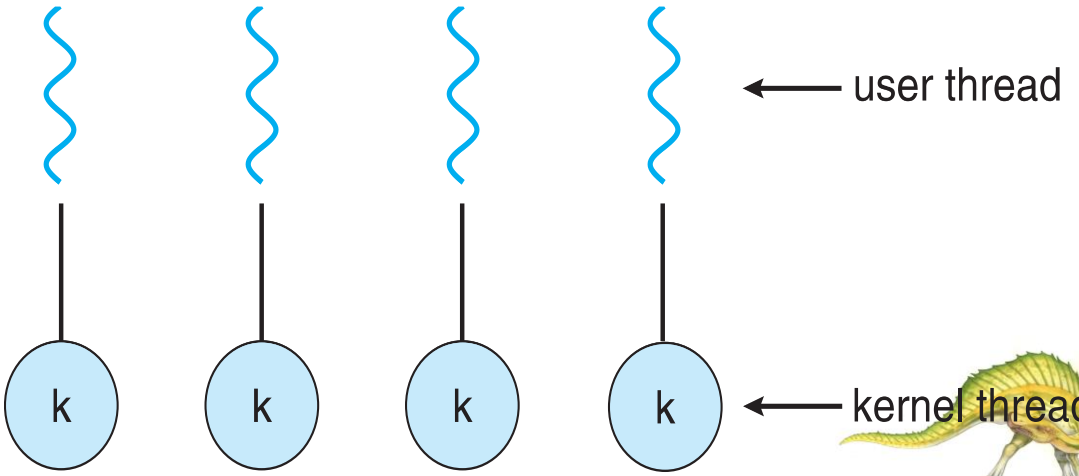
Many-to-One

- Many user-level threads mapped to single kernel thread
- The entire process will **block** if a thread makes a **blocking system call**.
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



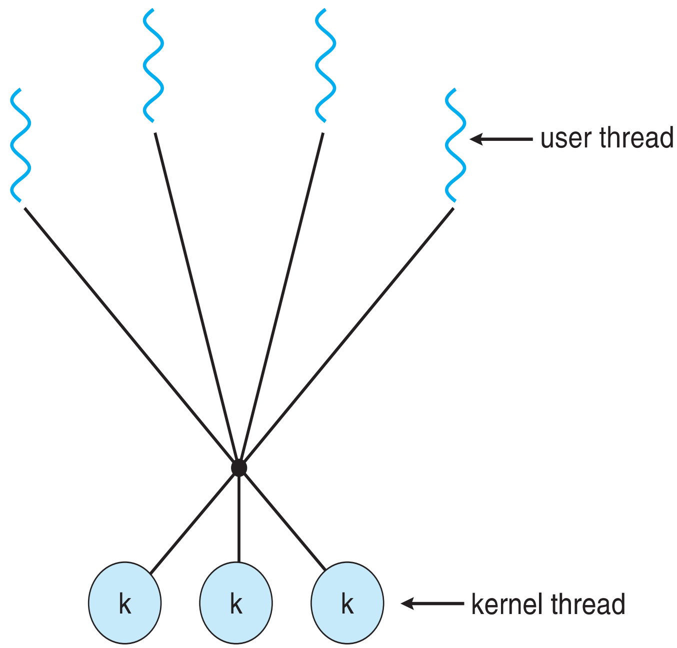
One-to-One

- Each user-level thread maps to kernel thread
- **Creating a user-level thread creates a kernel thread**
- **More concurrency than many-to-one**
  - It also allows multiple threads to run in parallel on multiprocessors
- 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
- When a thread performs a blocking system call, the **kernel can schedule another thread for execution**
- Solaris prior to version 9
- Windows with the *ThreadFiber* package



Thread Libraries

- For **POSIX** and Windows threading, any data declared **globally**—that is, declared outside of any function—are **shared** among all threads belonging to the same process.
- **Java** has **no notion of global data, access to shared data must be explicitly arranged between threads**
- Data declared local to a function are typically stored on the **stack**
- Since each thread has its own stack, each thread has its own copy of local data.

Asynchronous vs synchronous threading

- **Asynchronous threading**
  - Once the parent creates a child thread, the parent resumes its execution, so that the parent and child **execute concurrently**
  - Each thread runs independently of every other thread
    - E.g. Multithreaded web server
- **Synchronous threading**
  - Parent thread creates one or more children and then must wait for all of its children to terminate before it resumes
  - Known as **fork-join strategy**
  - **Threads can run concurrently but parent cannot continue until this work has been completed**
  - Synchronous threading involves **significant data sharing** among threads

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
  - ***It is a specification***, not an ***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)

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
- Two way of implementing Threads in Java
  - Extending Thread class
    - Create a new class that is derived from the Thread class and to override its run() method
  - Implementing the Runnable interface

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

    - When a class implements Runnable, it must define a run() method. The code implementing the run() method is what runs as a separate thread

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>"); }
}
```

Creating a Thread object does not specifically create the new thread; rather, the start() method creates the new thread

It calls the run() method, making the thread eligible to be run by the JVM.

(Note again that we never call the run() method directly.

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
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch
- Other methods include Intel's Threading Building Blocks (TBB), **`java.util.concurrent`** package

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**

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

Run for loop in parallel

```
#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;
}
```

Threading Issues

- **Issue:** The semantics of the **fork()** and **exec()** system calls **change** in a multithreaded program.
  - If one thread in a program calls **fork()**, does the new process duplicate all threads, or is the new process single-threaded?
- Signal handling
  - A **signal** is used in UNIX systems to notify a process that a particular **event** has **occurred**.
  - Synchronous and asynchronous
- Thread cancellation of target thread
  - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations

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);
```

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
- The functions pthread_key_create and pthread_key_delete are used respectively to create and delete a key for thread-specific data.
- The type of the key is explicitly left opaque and is referred to as pthread_key_t.

News in C11

- Multi-threading support (_Thread_local storage-class specifier, <threads.h> header including thread creation/management functions, mutex, condition variable and thread-specific storage functionality

News in C++11

- #include <threads.h>
- thread_local int foo = 0;
- C++11 introduces the thread_local keyword

End of Chapter 4