Operating Systems: Lecture 5

Threads & Concurrency

Jinwoo Kim
jwkim@jjay.cuny.edu

Chapter 4: Threads

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

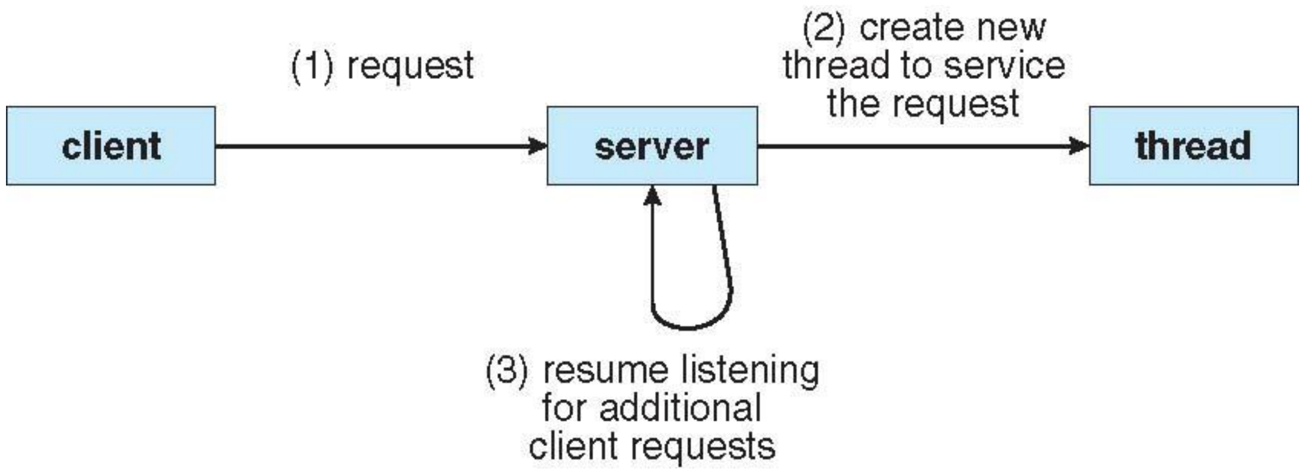
Objectives

- 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 incurs lower overhead than context switching
- Scalability
  - process can take advantage of multiprocessor architectures

What is Parallel Computing?

- In the simplest sense, ***parallel computing*** is the simultaneous use of multiple compute resources to solve a computational problem

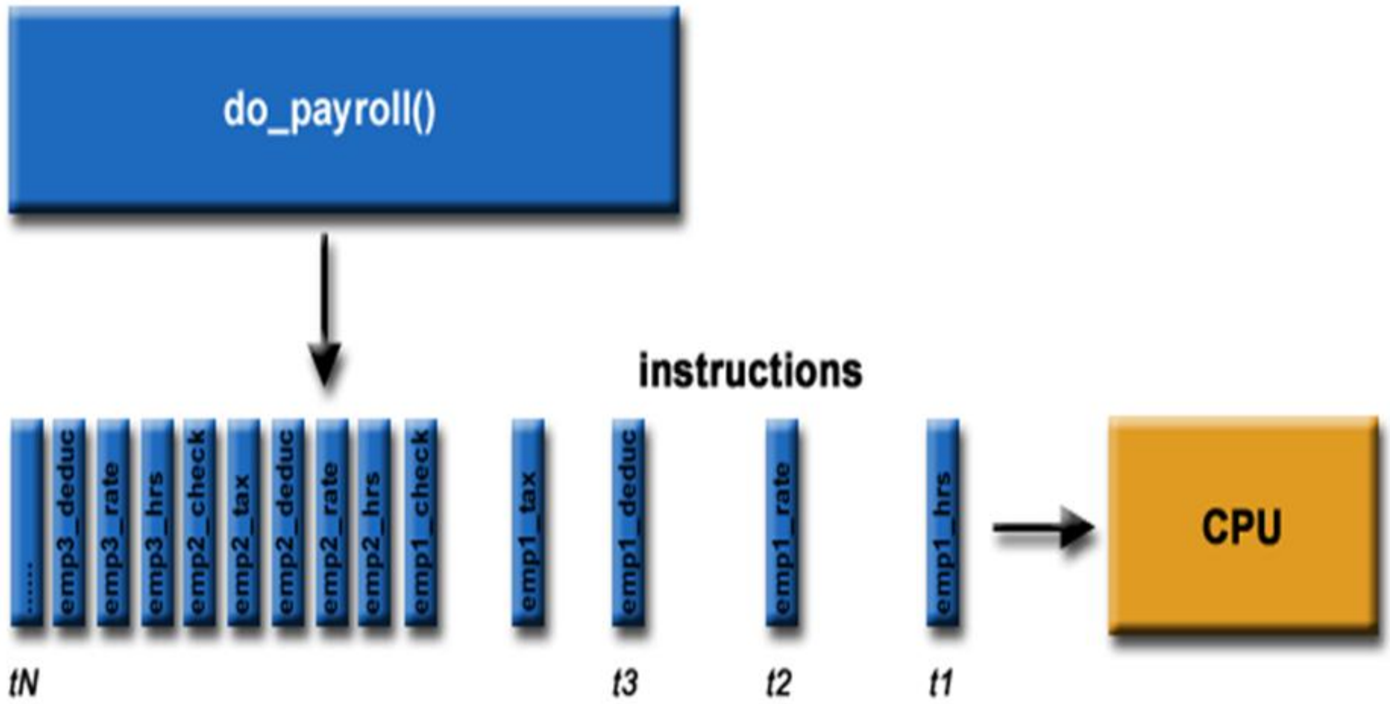
- Steps
  - A problem is broken into discrete parts that can be solved concurrently
  - Each part is further broken down to a series of instructions
  - Instructions from each part execute in sequence on each processor but simultaneously on different processors
  - An overall control/coordination mechanism is employed

What is Parallel Computing? (Continued)

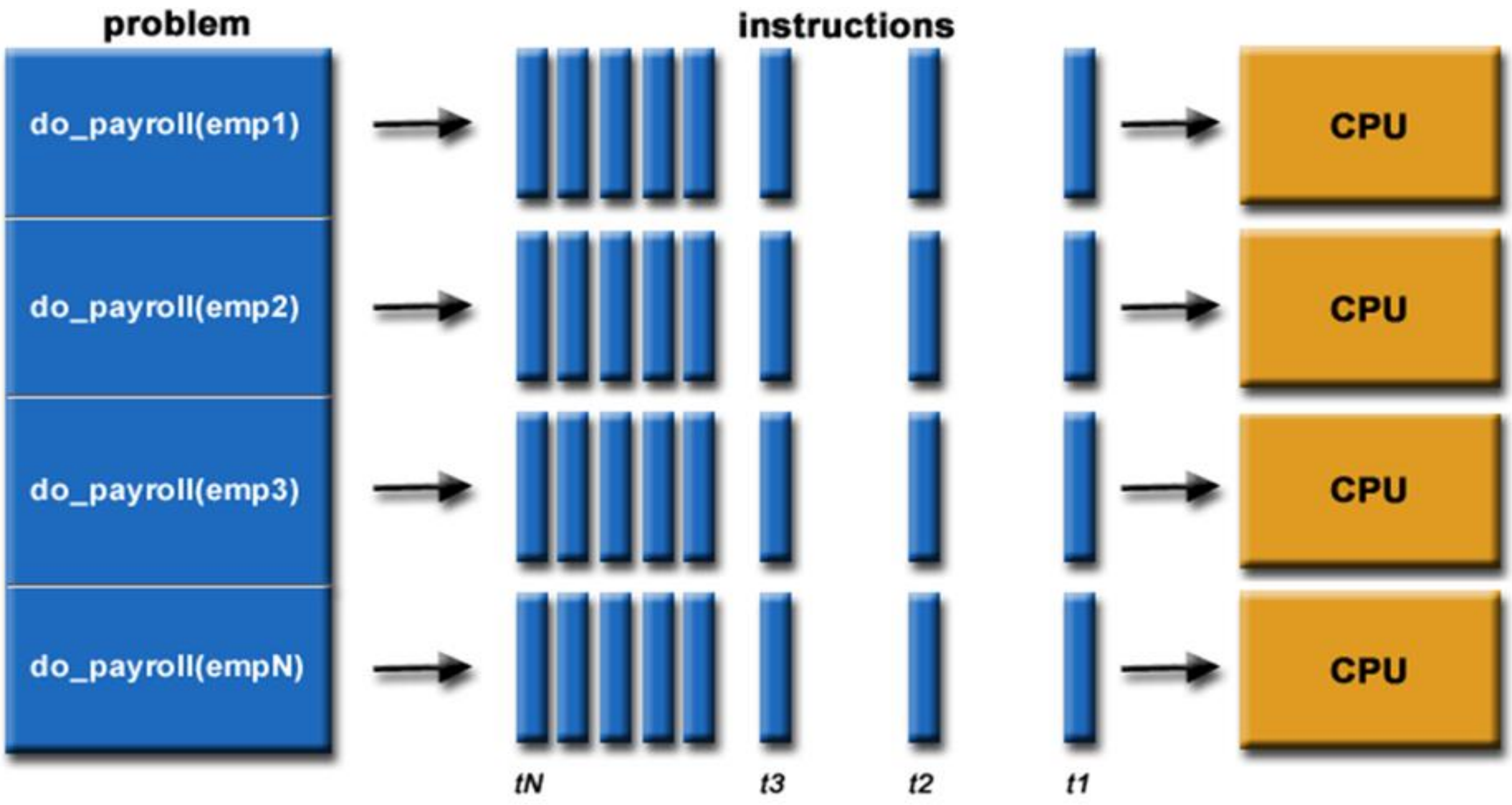
- The computational problem should be able to:
  - Be broken apart into discrete pieces of work that can be solved simultaneously
  - Execute multiple program instructions at any moment in time
  - Be solved in less time with multiple compute resources than with a single compute resource

- The compute resources might be:
  - A single computer with multiple processors
  - An arbitrary number of computers connected by a network
  - A combination of both

Sequential Computing

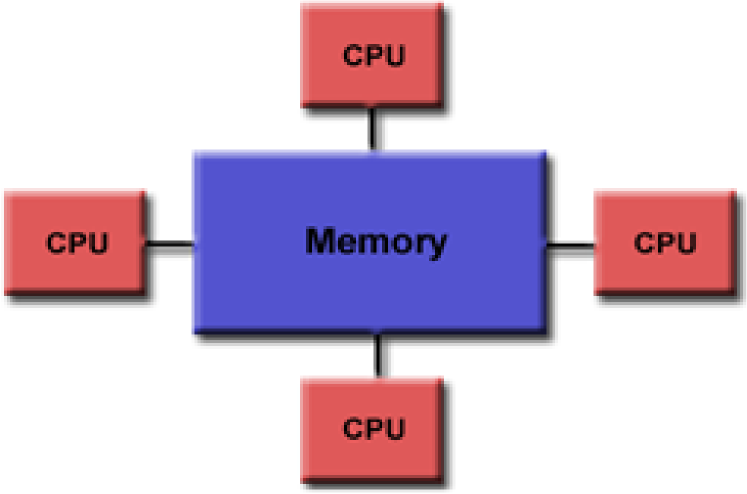


Parallel Computing



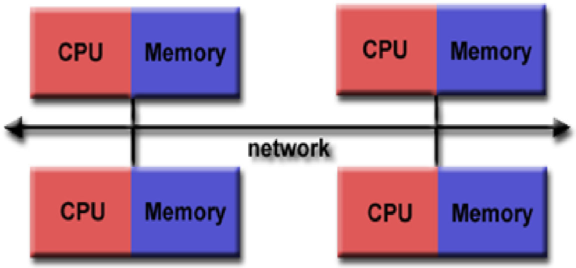
Shared Memory Architecture

- Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as ***global*** address space
- Multiple processors can operate independently but share the same memory resources
- Changes in a memory location affected by one processor are visible to all other processors
- The most critical problem to address is that of ***cache coherence***



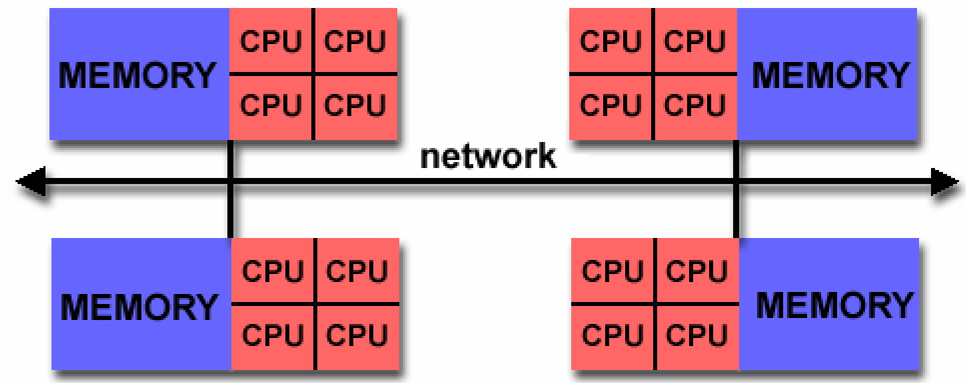
Distributed Memory Architecture

- Processors have their own local memory and runs their own copy of OS
  - Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors
- Like shared memory systems, distributed memory systems vary widely but share a common characteristic
  - Distributed memory systems require a communication network to connect inter-processor memory
- When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated
  - Synchronization between tasks is likewise the programmer's responsibility



Hybrid Architecture

- The large computers in the world today employ both shared and distributed memory architectures
- The shared memory component is usually a cache coherent SMP machine
  - Processors on a given SMP can address that machine's memory as global
- The distributed memory component is the networking of multiple SMPs. SMPs know only about their own memory
  - Not the memory on another SMP
  - Therefore, network communications are required to move data from one SMP to another



Accelerator-Based Architecture

- The largest and fastest computers in the world today employ Hybrid Architecture and Accelerators (GPUs)
  - Shared Memory
  - Distributed
  - Hybrid

History / Timeline

Decade	Parallel Hardware Platforms	Memory
1980s	Vector supercomputers	Shared
	Multiprocessors (networked)	Distributed
1990s	Cluster supercomputers	Distributed
	Internet	Distributed
	Symmetric multiprocessors	Shared
2000s	GPUs	Shared
	Multicore processors	Shared
2010s	Hybrid supercomputers/clusters	Both
	Coprocessors (w. vector units)	Shared

Fundamental Concepts in PDC

- Following are the PDC concepts that are pervasive irrespective of architecture, programming models, and tools
  - Asynchrony
  - Concurrency
  - Locality
  - Performance Measurement and Metric
  - Synchronization
  - Memory Hierarchy

Asynchrony

- Asynchrony is a characteristics of modern computers
  - Even though it seems like many operations are atomic, they are not
  - This is true for sequential computers too
- To develop parallel algorithms and applications we must understand the cause and effect of asynchrony and think about the mitigation
  - The mitigation often results into additional overhead
    - **Example: Data Race**

Concurrency

- Concurrency is a property of an algorithm, it exposes potential for parallelization
  - If concurrency is present in an algorithm then the concurrent operations can be executed in parallel (simultaneously) by multiple operation units (CPU's) if available
  - Without concurrency there is no scope for parallelization

- Concurrency can be present in a sequential program
  - parallelization takes advantage of concurrency to increase performance

Locality

- One of the overarching concepts in computing is that of locality of time, space, and state
- Each computational unit (a CPU in a shared memory machine or a node in a cluster) may have their own clock and their own notion of time
- Memory subsystems proactively predict and cache future memory references based upon recent memory reference patterns
- A challenge with localized control is the detection and management of conflict

Performance Measurement & Metric

- No matter what computing artifact (program algorithms, hardware) that we are designing, studying, and analyzing, we should be aware of how good the artifact is and strive to make it better

- Space, time, and energy are the basic commodities to measure and the metrics for these commodities may differ based on whether the context is sequential or parallel
  - For example in a sequential program run time may be used as a measure of goodness but in the parallel version of the same program there is an additional variable, the number of cores, so the notion of runtime does not capture the goodness

How do We Write Parallel Program: Types of Parallelism

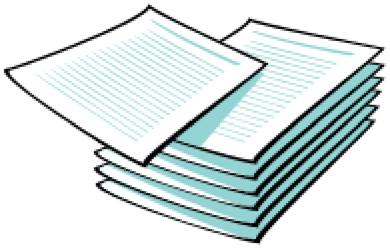
- Task Parallelism
  - Distributing threads across cores, each thread performing unique operation
    - **Example: two threads, each performing a unique statistical operation on the array of elements**
    - **The threads are operating in parallel on separate computing cores, but each is performing a unique operation**

- Data Parallelism
  - Distributes subsets of the same data across multiple cores, same operation on each
  - Each core carries out similar operations on its part of the data
    - **Example: summing the contents of an array of size N**
    - **For a single-core system, one thread would simply sum the elements [0] . . . [N − 1]**
    - **For a dual-core system, however, thread A, running on core 0, could sum the elements [0] . . . [N/2 − 1] and while thread B, running on core 1, could sum the elements [N/2] . . . [N − 1]**
    - **So the Two threads would be running in parallel on separate computing cores**

Professor S(erial)

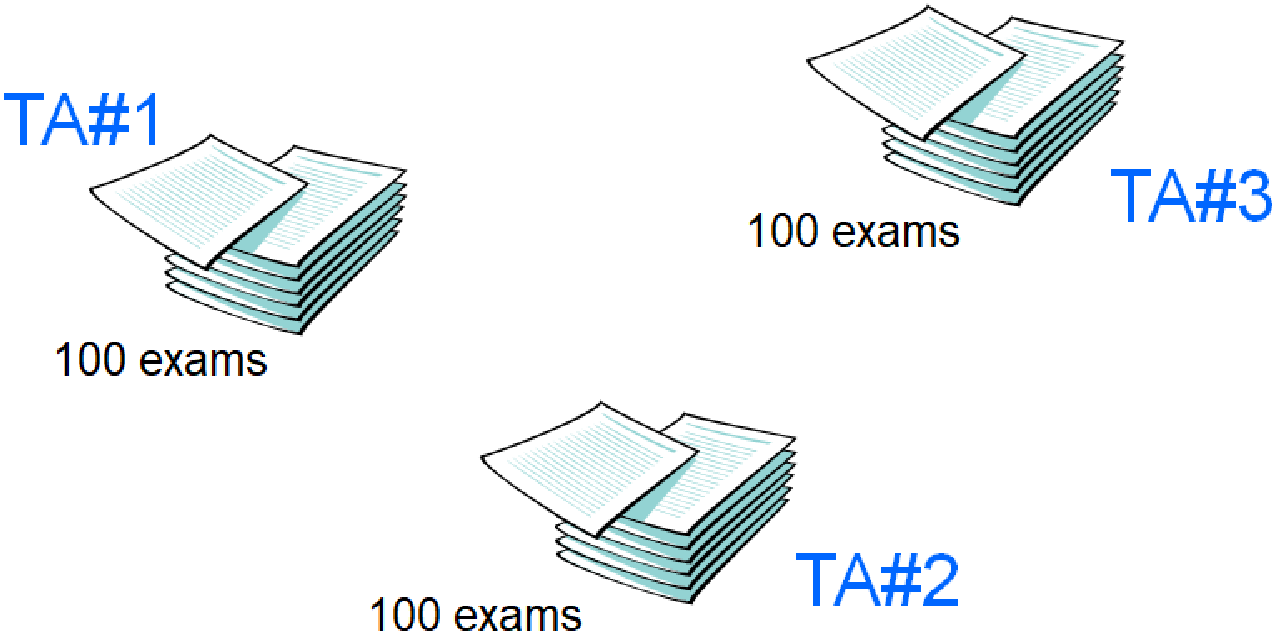
15 questions

300 exams



Professor S's Teaching Assistants

Division of work – Data or Task parallelism ???



Division of work – Data or Task parallelism ???

TA#1

Questions 1 - 5

TA#3

Questions 11 - 15

Questions 6 - 10

TA#2

Multicore Programming

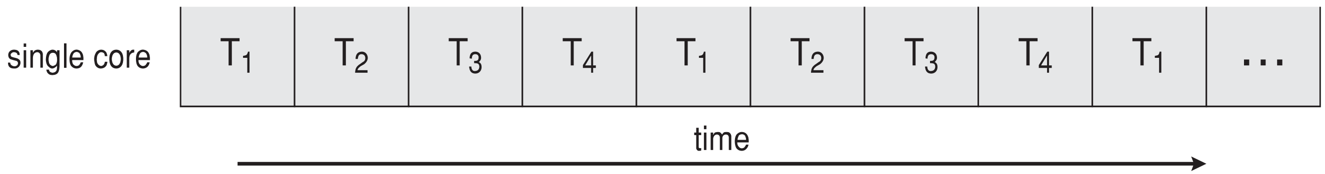
- Multicore or multiprocessor systems putting pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging

- *Concurrency* supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

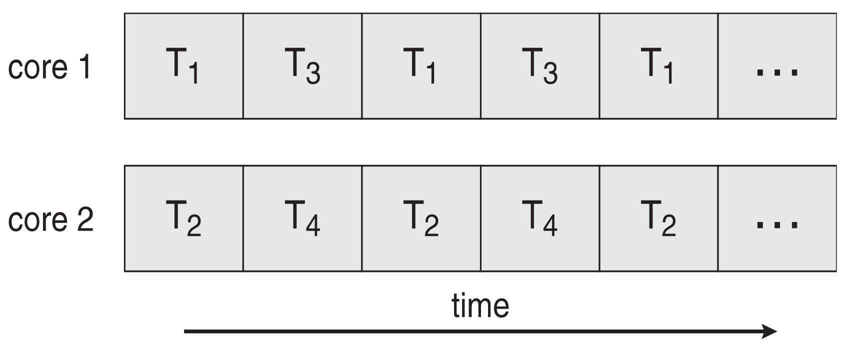
- *Parallelism* implies a system can perform more than one task simultaneously

Concurrency vs. Parallelism

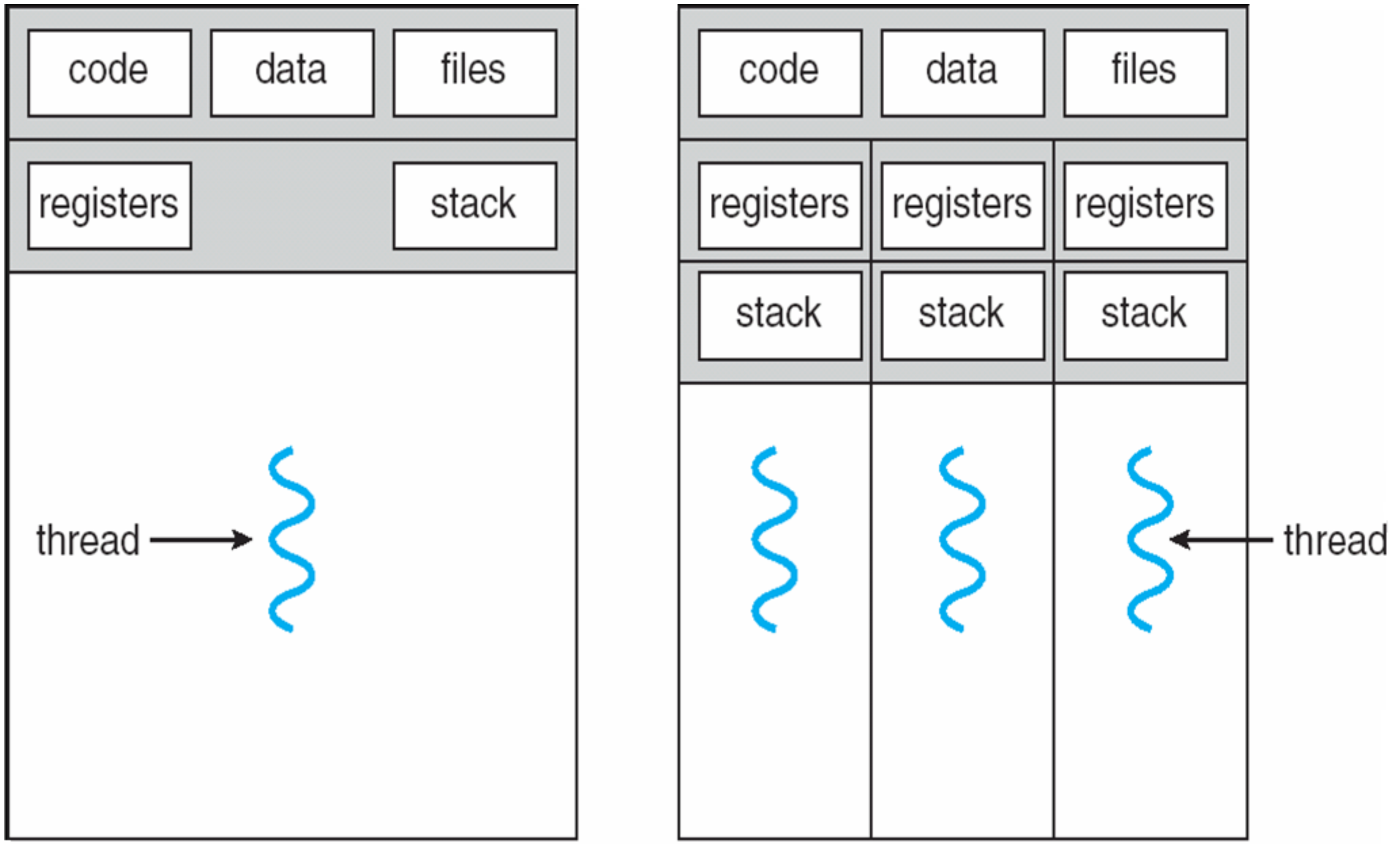
Concurrent execution on single-core system:



Parallelism on a multi-core system:



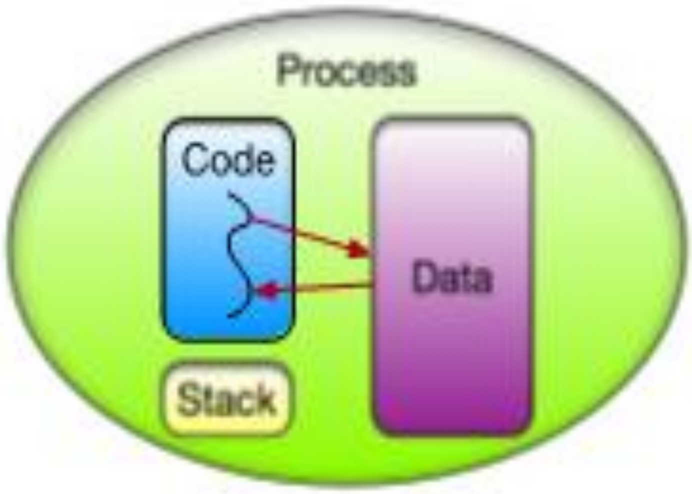
Single and Multithreaded Processes



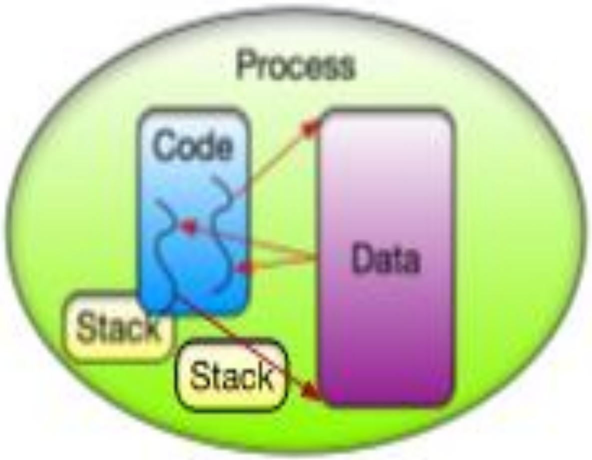
single-threaded process

multithreaded process

Process vs Thread



Process with one thread



Process with two threads

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

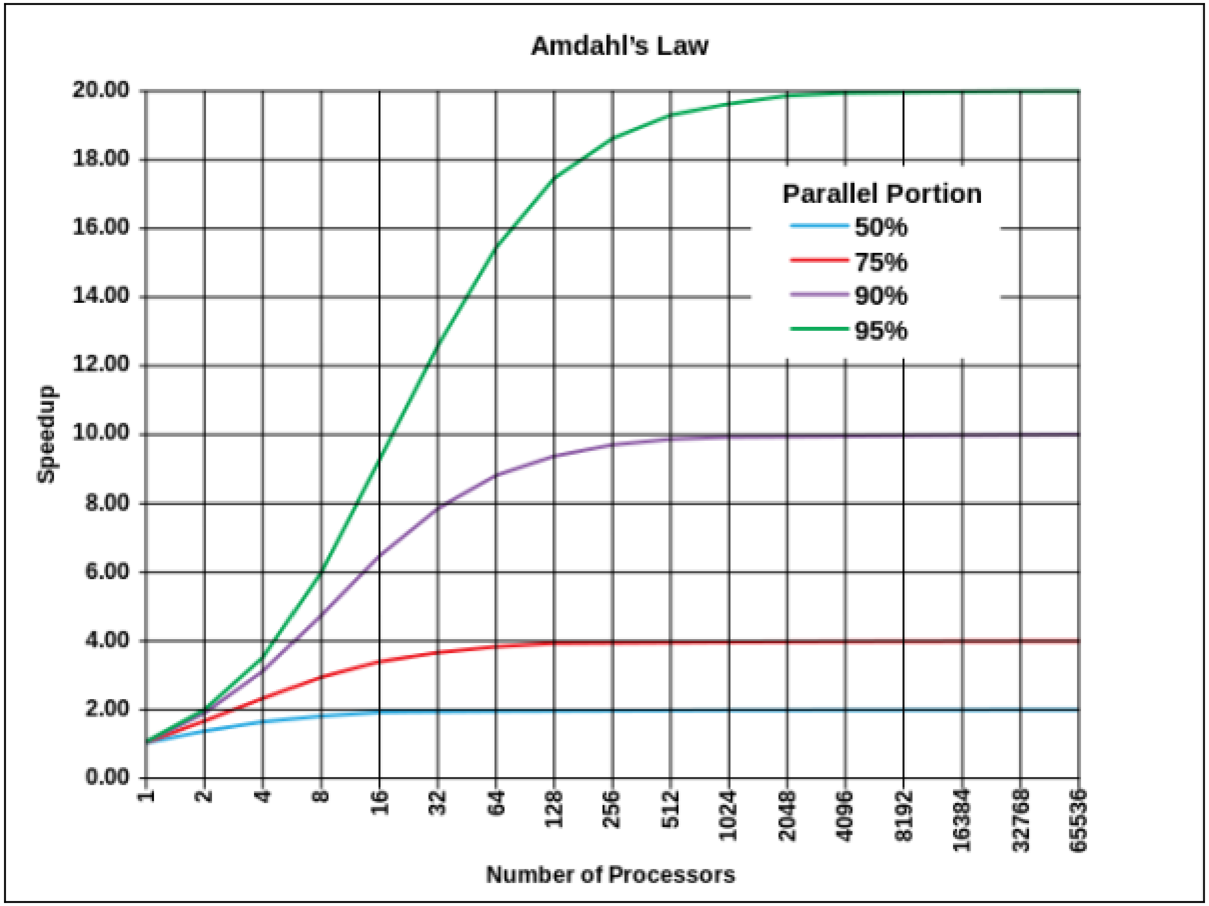
$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- 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 consider contemporary multicore systems?

Amdahl's Law (Continued)



- Figure from: https://en.wikipedia.org/wiki/Parallel_computing

User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads

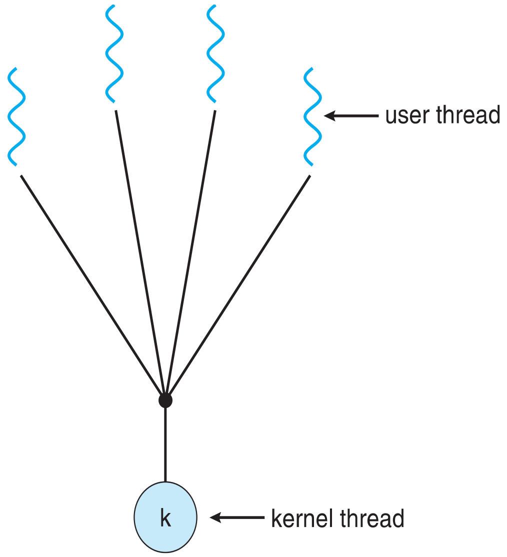
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general-purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Multithreading Models

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

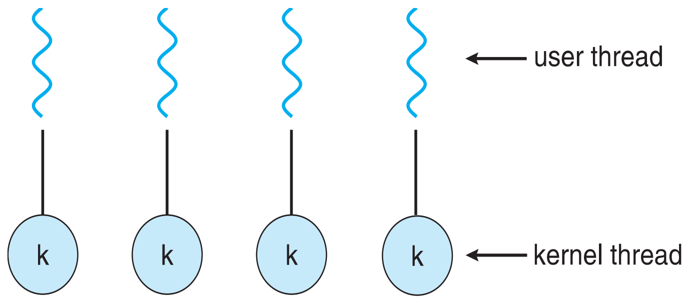
Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on 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
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows NT and later
  - Linux
  - Solaris 9 and later



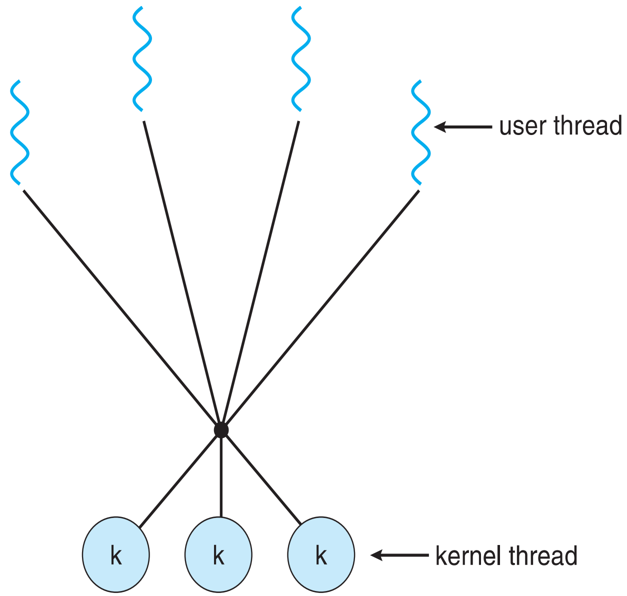
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads

- Allows the operating system to create enough kernel threads

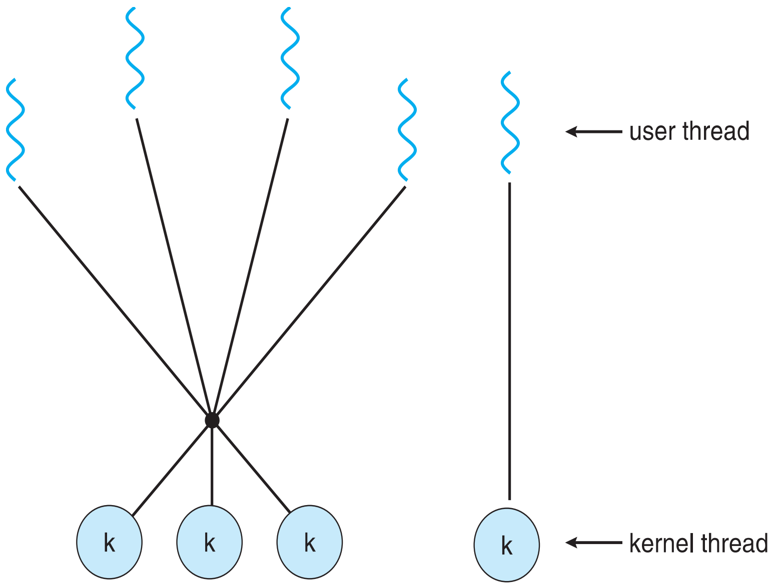
- Solaris prior to version 9

- 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

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.
    */
}
```

Introduction to OpenMP

- An Application Program Interface (API) that may be used to explicitly direct ***multi-threaded, shared memory parallelism***

- Comprised of three primary API components
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables

- An abbreviation for
  - Short version: **Open Multi-Processing**
  - Long version: **Open** specifications for **Multi-Processing** via collaborative work between interested parties from the hardware and software industry, government and academia

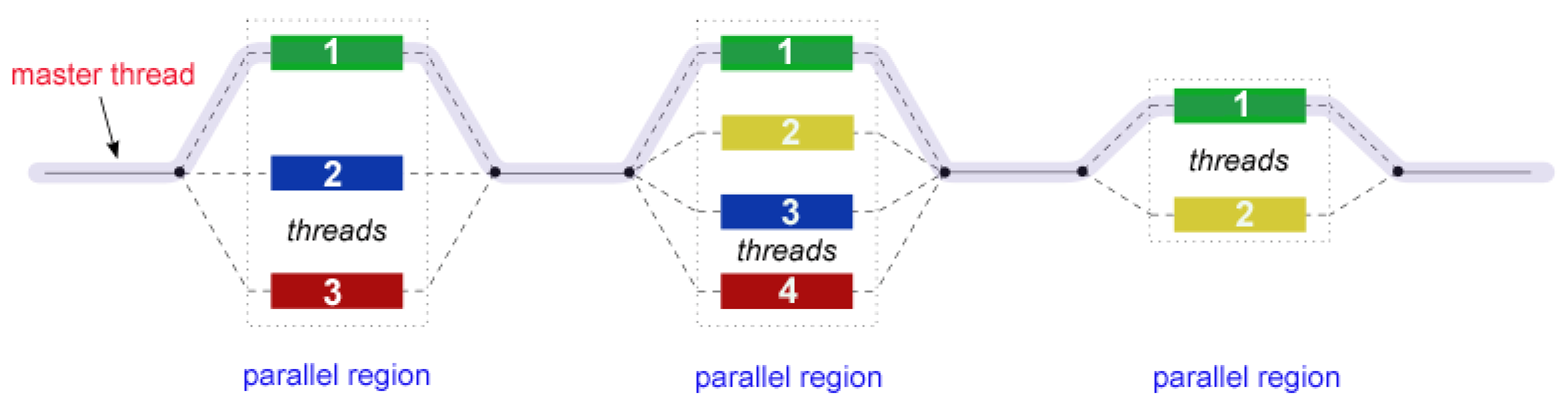
OpenMP is Not

- Meant for distributed memory parallel systems (by itself)
- Necessarily implemented identically by all vendors
- Guaranteed to make the most efficient use of shared memory
- Required to check for data dependencies, data conflicts, race conditions, or deadlocks
- Required to check for code sequences that cause a program to be classified as non-conforming
- Meant to cover compiler-generated automatic parallelization and directives to the compiler to assist such parallelization
- Designed to guarantee that input or output to the same file is synchronous when executed in parallel
  - The programmer is responsible for synchronizing input and output

OpenMP Programming Model

Multithreading: a master thread forks a specified number of slave threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors (or cores).

Fork Join Model



Three Components

- The OpenMP API is comprised of three distinct components.
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables

- The application developer decides how to employ these components

- Implementations differ in their support of all API components
  - For example, an implementation may state that it supports nested parallelism, but the API makes it clear that may be limited to a single thread - the master thread. Not exactly what the developer might expect?

Compiler Directives

- OpenMP compiler directives are used for various purposes
  - Spawning a parallel region
  - Dividing blocks of code among threads
  - Distributing loop iterations between threads
  - Serializing sections of code
  - Synchronization of work among threads

- Compiler directives have the following syntax:

  sentinel directive-name [clause, ...]

  - **Directive-name** is a specific keyword, for example parallel, that defines and controls the action(s) taken
  - **Clauses**, for example private, can be used to further specify the behavior
  - # pragma omp parallel num_threads(thread_count)
  - # pragma omp parallel default(shared) private(beta,pi)

Compile and Run OpenMP programs

- Compile C/C++ codes
  - \> gcc/g++ -fopenmp name.c -o name
  - \> icc/icpc -openmp name.c -o name

- Run OpenMP programs
  - \> export OMP_NUM_THREADS=4 # set number of threads
  - \> ./name
  - \> time ./name # run and measure the time.

Runtime Routines

- The OpenMP API includes an ever-growing number of run-time library routines

- These routines are used for a variety of purposes:
  - Setting and querying the number of threads
  - Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, the thread team size
  - Setting and querying the dynamic threads feature
  - Querying if in a parallel region, and at what level
  - Setting and querying nested parallelism
  - Setting, initializing and terminating locks and nested locks
  - Querying wall clock time and resolution

Environment Variable

- OpenMP provides several environment variables for controlling the execution of parallel code at run-time

- These environment variables can be used to control such things as:
  - Setting the number of threads
  - Specifying how loop iterations are divided
  - Binding threads to processors
  - Enabling/disabling nested parallelism; setting the maximum levels of nested parallelism
  - Enabling/disabling dynamic threads
  - Setting thread stack size

OpenMP Thread Model

- Program start with one thread(Master)

- Before parallel region
  - Multiple threads are created
  - Threads have id (0 to p-1)
    - **master thread id is 0**

- At the end of parallel region thread 1 to p-1 join with the thread 0

- There can be multiple parallel region

OpenMP Code Structure

#include <omp.h>

main ()

{

int var1, var2, var3;

Serial code . . .

Beginning of parallel section. Fork a team of threads. Specify variable scoping

#pragma omp parallel private(var1, var2) shared(var3)

{

Run-time Library calls .

Other OpenMP directives .

} *At the end of a parallel region, there is an implied barrier that forces all threads to wait until the computation inside the region has been completed.*

Resume serial code . . .

}

Pragmas

- Special preprocessor instructions
- Typically added to a system to allow behaviors that aren't part of the basic C specification
- Compilers that don't support the pragmas ignore #pragma
- OpenMP directives have three parts
  - #pragma omp, Directive, Optional clause (modifies directive)

    Example:

    ```
    # pragma omp  parallel num_thread(6)
    ```

    Directive → parallel; Clause → num_thread(6)

Parallel Directive

pragma omp parallel

- Most basic parallel directive
  - Creates multiple thread
  - Following structured block of code(parallel region) is executed by the threads parallelly (asynchronously)

- There is an implicit barrier at the end of a parallel region

How Many Threads

- Determined by ***num_thread*** clause (more in next slide)

- If ***num_thread*** clause is absent, the number of thread is determined by the value OMP_NUM_THREADS environment variable

- If the variable is not set by the user, then the number of thread is system depended (usually equal to the number of cores- including hyperthreading)

num_thread Clause

- Tells OpenMP runtime systems how many threads to create
- #pragma omp parallel num_thread(8)
  - will create seven new thread (total 8 including master)
- #pragma omp parallel num_thread(x)
  - x must be an integer expression with the runtime value >=1
  - often pass as command line argument

Restriction on Parallel Region

- A parallel region must be a structured block that does not span multiple routines or code files
- It is illegal to branch (including **goto**) into or out of a parallel region
- Only a single **IF** clause is permitted
- Only a single **NUM_THREADS** clause is permitted

Runtime Functions

- Following are two most used runtime functions

- int omp_get_num_threads()

  returns *<u>number of threads</u>* in the team

- int omp_get_thread_num()

  returns *<u>thread id</u>* (between 0 to p-1) of thread which called this function

Hello World Revisited

```
#include <iostream>
#include <omp.h>
int main()
{
      std::cout << “Hello World!\n”;
     #pragma omp parallel
     {
int thread_count = omp_get__num_threads();
int my_id = omp_get_thread_num();
std::cout << “Hello World from ” << my_id << “ of ” << thread_count
<< std::endl;
    }
     return 0;
}
```

Parallel Directive Clauses

#pragma omp parallel *[clause ...] newline*

num_threads *(integer-expression)*

private *(list)*

shared *(list)*

default (shared | none)

defines variable scope

reduction *(operator: list)*

firstprivate *(list)*

lastprivate(list)

copying*(list)*

If *(scalar_expression)*

and more ……...

will discuss later

Scope of Variable

- In serial programming, the scope of a variable consists of those *parts of a program in which the variable can be used*

- In OpenMP, the scope of a variable refers to the *set of threads* that can access the variable in a *parallel block*

Variable Scope in OpenMP

- A variable that can be accessed by all the threads in the team has **shared** scope

- A variable that can only be accessed by a single thread has **private** scope

- The **default** scope for variables declared *before* a parallel block is **shared** and variable declared *in* the block are **private**

- The default scope can be changed by default, private and shared clause

#pragma omp parallel default(shared) private(beta,pi)

Data Scope Clauses

- Private
  - A new object of the same type is declared once for each thread in the team
  - All references to the original object are replaced with references to the new object
  - Variables declared **PRIVATE** should be assumed to be uninitialized for each thread

- Shared
  - The SHARED clause declares variables in its list to be shared among all threads in the team
  - It is the programmer's responsibility to ensure that multiple threads properly access **SHARED** variables (such as via **CRITICAL** sections)

Default Data Scope Clauses

- The **DEFAULT** clause allows the user to specify a default scope for all variables in the lexical extent of any parallel region

- Using **NONE** as a default requires that the programmer explicitly scope all variables

- Only one **DEFAULT** clause can be specified on a PARALLEL directive

Data Scope Example

```
#include <iostream>
#include <omp.h>
int main(int argc, int *argv[]){
      int p, total = 0;
      std::cout << "Enter the number of threads you want to run: ";
      std::cin >> p;
      #pragma omp parallel num_threads(p)
      {   int x = 0;
    int thread_count = omp_get_num_threads();
    int my_id = omp_get_thread_num();
x = (my_id + 1) * thread_count;
total = total + x;
std::cout << “Id = ” << my_id << “ x = ” << x << std::endl;
    }
      std::cout << “Total= ” << total << std::endl;
      return 0;
}
```

Synchronization with CRITICAL Directive

- Private
  - The CRITICAL directive specifies a region of code that must be executed by only one thread at a time
    - **If a thread is currently executing inside a CRITICAL region and another thread reaches that CRITICAL region and attempts to execute it, it will block until the first thread exits that CRITICAL region.**
    - **It is illegal to branch into or out of a CRITICAL block**

```
#pragma omp parallel shared(x)
{
   #pragma omp critical
    x = x + 1;
} /* end of parallel section */
```

Synchronization Example

```
#include <iostream>
#include <omp.h>
int main(int argc, int *argv[]){
     int p, total = 0;
     std::cout << "Enter the number of threads you want to run: ";
     std::cin >> p;
     #pragma omp parallel num_threads(p)
     {  int x = 0;
   int thread_count = omp_get_num_threads();
   int my_id = omp_get_thread_num();
   x = (my_id+1)*thread_count;
   #pragma omp critical
       total = total + x;
   std::cout << “Id = ” << my_id << “ x = ” << x << std::endl;
     }
     std::cout << “Total= ” << total << std::endl;
     return 0;
}
```

Divide for-loop for parallel sections

```
for (int i=0; i<8; i++) x[i]=0; //run on 4 threads
```

```
#pragma omp parallel
        {
        int numt=omp_get_num_thread();
        int id = omp_get_thread_num(); //id=0, 1, 2, or 3
        for (int i=id; i<8; i +=numt)
                                x[i]=0;

        }
```

// Assume number of threads=4

Thread 0	Thread 1	Thread 2	Thread 3
Id=0;	Id=1;	Id=2;	Id=3;
x[0]=0;	x[1]=0;	x[2]=0;	x[3]=0;
X[4]=0;	X[5]=0;	X[6]=0;	X[7]=0;

Use pragma parallel for

```
for (int i=0; i<8; i++) x[i]=0;
```

```
#pragma omp parallel for
{
            for (int i=0; i<8; i++)
                        x[i]=0;
}
```

System divides loop iterations to threads

Id=0; **x[0]=0;** X[4]=0;	**Id=1;** **x[1]=0;** X[5]=0;	**Id=2;** **x[2]=0;** X[6]=0;	**Id=3;** **x[3]=0;** X[7]=0;

OpenMP Data Parallel Construct: Parallel Loop

- Compiler calculates loop bounds for each thread directly from *serial* source (computation decomposition)
- Compiler also manages data partitioning
- Synchronization also automatic (barrier)

Serial Program:	Parallel Program:

Serial Program:

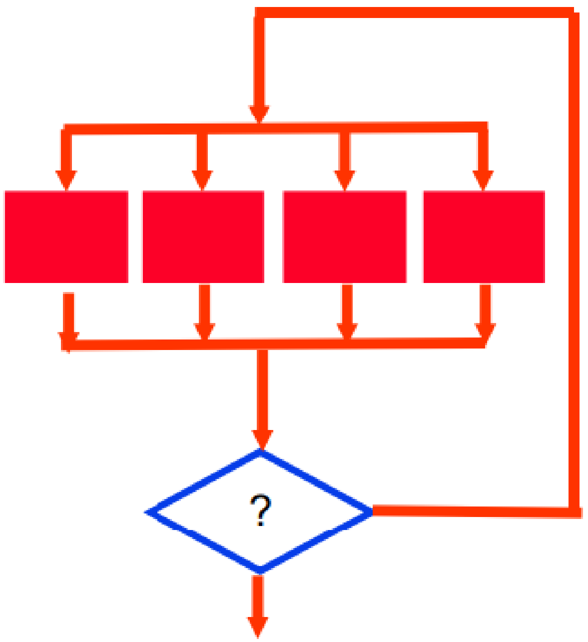
```
void main()
{
   double Res[1000];

   for(int i=0;i<1000;i++) {
          do_huge_comp(Res[i]);
   }
}
```

Parallel Program:

```
void main()
{
   double Res[1000];
#pragma omp parallel for
   for(int i=0;i<1000;i++) {
          do_huge_comp(Res[i]);
   }
}
```

Programming Model – Parallel Loops

- Requirement for parallel loops
  - No data dependencies (reads/write or write/write pairs) between iterations!
- Preprocessor calculates loop bounds and divide iterations among parallel threads




```
#pragma omp parallel for

 for( i=0; i < 25; i++ )
 {
   printf("Foo");
 }
```

Be Careful with Data Dependences

Whenever a statement in a program reads or writes a memory location and another statement reads or writes the same memory location, and at least one of the two statements writes the location, then there is a data dependence on that memory location between the two statements. The loop may not be executed in parallel.

```
for(i=1;i< N; i++)
{
   a[i] = a[i] + a[i-1];
}
```

a[i] is written in loop iteration i and read in loop iteration i+1. This loop can not be executed in parallel. The results may not be correct.

Example

```
for (i=0; i<max; i++) zero[i] = 0;
```

- Breaks *for loop* into chunks, and allocate each to a separate thread
  - e.g. if `max` = 100 with 2 threads:
    assign 0-49 to thread 0, and 50-99 to thread 1
- Must have relatively simple “shape” for an OpenMP-aware compiler to be able to parallelize it
  - Necessary for the run-time system to be able to determine how many of the loop iterations to assign to each thread
- No premature exits from the loop allowed
  - i.e. No `break`, `return`, `exit`, `goto` statements

In general, don’t jump outside of any pragma block

Example

- Distribute iterations in a parallel region

```
#pragma omp parallel for shared(n,a) private(i)
   for (i=0; i<n; i++)
        a[i] = i + n;
```

- shared clause: All threads can read from and write to a shared variable.
- private clause: Each thread has a local copy of a private variable.
- The maximum iteration number *n* is shared, while the iteration number *i* is private.
- Each thread executes a subset of the total iteration space i = 0, . . . , n − 1
- The mapping between iterations and threads can be controlled by the schedule clause.

Example

- Two work-sharing loops in one parallel region

```
#pragma omp parallel shared(n,a,b) private(i)
{
   #pragma omp for
   for (i=0; i<n; i++)   a[i] = i+1;
   //  there is an implied barrier

   #pragma omp for
   for (i=0; i<n; i++)   b[i] = 2 * a[i];
}    /*-- End of parallel region --*/
```

- The distribution of iterations to threads could be different for the two loops.
- The implied barrier at the end of the first loop ensures that all the values of a[i] are updated before they are used in the second loop.

Data race condition

- Data race conditions arise when multithreads read or write the same shared data simultaneously.
- Example: two threads each increases the value of a shared integer variable by one.

Correct sequence

Thread 1	Thread 2		value
			0
read value		←	0
Increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

Incorrect sequence

Thread 1	Thread 2		value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

Data race condition (Continued)

- Example of data racing: sums up elements of a vector

Different threads read and write the shared data sum simultaneously.

A data race condition arises!

The final result of *sum* could be incorrect!

```
sum = 0;
 #pragma omp parallel for shared(sum,a,n) private(i)
  for (i=0; i<n; i++)
  {
     sum = sum + a[i];
  }  /*-- End of parallel for --*/
printf("Value of sum after parallel region: %f\n",sum);
```

Atomic construct

- The atomic construct allows multiple threads to safely update a shared variable.
- The memory update (such as write) in the next instruction will be performed atomically. It does not make the entire statement atomic. Only the memory update is atomic.
- It is applied only to the (single) assignment statement that immediately follows it.

	C/C++ programs	Fortran programs
• Syntax	**#pragma omp atomic** ...... a single statement	**!$omp atomic** ...... a single statement ..... **!$omp end atomic**
• Supported operators	+, *, -, /, &, ^, |, <<, >>.	+, *, -, /, .AND., .OR., .EQV., .NEQV. .

Atomic construct (Continued)

- The first try to solve the data-race problem: use *atomic* (correct but slow)
- The atomic construct avoids the data racing condition. Therefore the result is correct.
- But all elements are added sequentially, and there is performance penalty for using *atomic*, because the system coordinates all threads.
- This code is even slower than a serial code!

```
sum = 0;
 #pragma omp parallel for shared(n,a,sum) private(i) // Optimization: use reduction instead of atomic
 for (i=0; i<n; i++)
 {
      #pragma omp atomic
       sum += a[i];
 }  /*-- End of parallel for --*/
printf("Value of sum after parallel region: %d\n",sum);
```

Atomic construct (Continued)

- A partially parallel scheme to avoid data race

Step 1: Calculate local sums in parallel

Thread 1	Thread 2	......	Thread m
a_0	a_m	......	a_{n-m-1}
+	+		+
a_1	a_{m+1}	......	a_{n-m}
+	+		+
⋮	⋮		⋮
+	+		+
a_{m-1}	a_{2m-1}	......	a_n
=	=		=
LS_1	LS_2	......	LS_m

m: number of threads

n: array length

LS: local sum

Atomic construct (Continued)

Step 2: Update total sum sequentially

Thread 1	Thread 2	……	Thread m
Read initial S			
$S = S + LS_1$			
Write S			
	Read S		
	$S = S + LS_2$		
	Write S		
		……	
			Read S
			$S = S + LS_m$
			Write S

m: number of threads

LS: local sum

S: total sum

Atomic construct (Continued)

- The second try to solve the data-race problem: use *atomic* (correct and fast)

- Each thread adds up its local sum.
- The *atomic* is only applied for adding up local sums to obtain the total sum.

```
sum = 0;
#pragma omp parallel shared(n,a,sum) private(sumLocal)
{
    sumLocal = 0;
    #pragma omp for
    for (i=0; i<n; i++)  sumLocal += a[i];
    #pragma omp atomic
    sum += sumLocal;
}  /*-- End of parallel region --*/
printf("Value of sum after parallel region: %d\n",sum);
```

Critical construct

- The critical construct provides a means to ensure that multiple threads do not attempt to update the same shared data simultaneously.
- The enclosed code block will be executed by only one thread at a time.
- When a thread encounters a critical construct, it waits until no other thread is executing a critical region with the same name.

- Syntax in C/C++ programs

```
#pragma omp critical [(name)]
...... code block .....
```

- Syntax in Fortran programs

```
!$omp critical [(name)]
...... code block .....
!$omp end critical [(name)]
```

Critical construct (Continued)

- The third try to solve the data-race problem: use *critical* (correct and fast)
- Each thread adds up its local sum.
- The critical region is used to avoid a data race condition when updating the total sum.

```
sum = 0;
#pragma omp parallel shared(n,a,sum) private(sumLocal)
{
    sumLocal = 0;
    #pragma omp for
    for (i=0; i<n; i++)  sumLocal += a[i];
    #pragma omp critical (update_sum)
    {
      sum += sumLocal;
      printf("TID=%d: sumLocal=%d sum = %d\n", omp_get_thread_num(), sumLocal, sum);
    }
}  /*-- End of parallel region --*/
printf("Value of sum after parallel region: %d\n",sum);
```

Critical construct (Continued)

- Another example of critical construct: avoid garbled output

A critical region helps to avoid intermingled output when multiple threads print from within a parallel region.

```
#pragma omp parallel private(TID)
{
  TID = omp_get_thread_num();
  #pragma omp critical (print_tid)
  {
      printf("Thread %d : Hello, ",TID);
      printf("world!\n");
  }
} /*-- End of parallel region --*/
```

Reduction construct

- The fourth try to solve the data-race problem: use *reduction* (correct, fast and simple)

```
#pragma omp parallel for default(none) shared(n,a) private(i) reduction(+:sum)
 for (i=0; i<n; i++)
        sum += a[i];
 /*-- End of parallel reduction --*/
```

- The reduction variable is protected to avoid data race.
- The partially parallel scheme mentioned before is applied behind the scene.
- An OpenMP compiler will generate a roughly identical machine code for using reduction clause (the code in this page) and for using critical construct (the code in a previous page).
- The reduction variable is shared by default and it is not necessary to specify it explicitly as "shared".

Reduction construct (Continued)

- Operators and statements supported by the reduction clause

	C/C++	Fortran
Typical statements	x = x *op* expr x *binop* = expr x = expr *op* x (except for subtraction) x++ ++x x-- --x	x = x *op* expr x = expr *op* x (except for subtraction) x = *intrinsic* (x, expr_list) x = *intrinsic* (expr_list, x)
op could be	+, *, -, &, ^, |, &&, or ||	+, *, -, .and., .or., .eqv., or .neqv.
binop could be	+, *, -, &, ^, or |	N/A
Intrinsic function could be	N/A	max, min, iand, ior, ieor

Project #2

For this assignment you need to write a parallel program in C++ using OpenMP for vector addition. Assume A, B, C are three vectors of equal length. The program will add the corresponding elements of vectors A and B and will store the sum in the corresponding elements in vector C (in other words C[i] = A[i] + B[i]). Every thread should execute approximately equal number of loop iterations. The only OpenMP directive you are allowed to use is:

```
#pragma omp parallel num_threads(no of threads)
```

The program should take n and the number of threads to use as command line arguments:

```
./parallel_vector_addition <n> <threads>
```

Where ***n*** is the <u>length of the vectors</u> and ***threads*** is the <u>number of threads to be created</u>.

Pseudocode for Assignment

```
mystart = myid*n/p;   // starting index for the individual thread
myend = mystart+n/p;  // ending index for the individual thread
for (i = mystart; i < myend; i++)  // each thread computes local sum
do vector addition               // and later all local sums combined
```

Project #2 (Continued)

As an input vector A, initialize its size to 10,000 and elements from 1 to 10,000.

So, A[0] = 1, A[1] = 2, A[2] = 3, … , A[9999] = 10000.

Input vector B will be initialized to the same size with opposite inputs.

So, B[0] = 10000, B[1] = 9999, B[2] = 9998, … , B[9999] = 1

Using above input vectors A and B, create output Vector C which will be computed as

C[i] = A[i] + B[i];

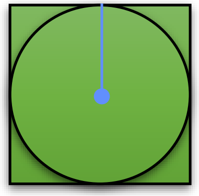
You should check whether your output vector value is 10001 in every C[i].

First, start with 2 threads (each thread adding 5,000 vectors), and then do with 4,and and 8 threads. Remember sometimes your vector size can not be divided equally by number of threads. You need to slightly modify pseudo code to handle the situation accordingly. (Hint: If you have p threads, first (p - 1) threads should have equal number of input size and the last thread will take care of whatever the remainder portion.) Check the running time from each experiment and compare the result. Report your findings from this project in a separate paragraph.

Your output should show team of treads do evenly distributed work, but big vector size might cause an issue in output. You can create mini version of original vector in much smaller size of 100 (A[0] = 1, A[1] = 2, A[2] = 3, … , A[99] = 100) and run with 6 threads once and take a snap shop of your output. And run with original size with 2, 4, and 8 threads to compare running times.

Slightly More Complex Example

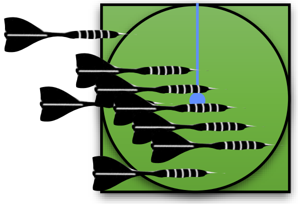
- **Problem: Estimate Pi**



- Consider a circle inside of a square
- Let *p* be the ratio of the area of the circle to the area of the square, then

$$p = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

- So: $\pi = 4p$

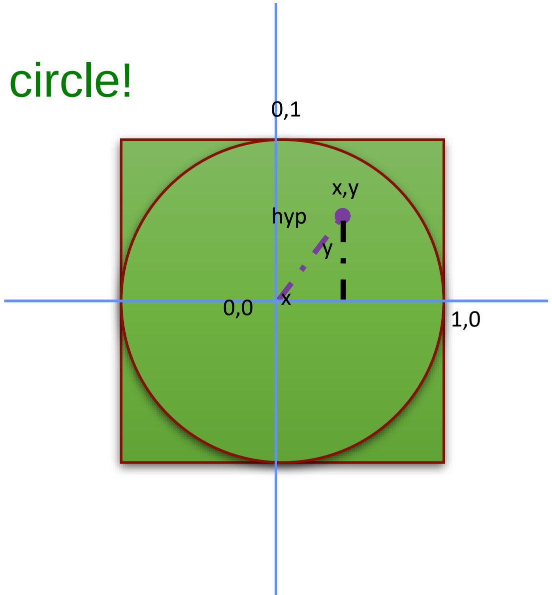


- How do we figure out *p*? The **Monte Carlo** Method
- Throw darts at the square. Lots and lots of darts.
- Count the number of darts that land inside the circle
- Divide the number of darts that land in the circle by total number thrown to estimate *p* !!!
- Multiply by 4, and we have an estimate for π

Some Unresolved Questions

- How does a computer throw darts?
  - By generating random x,y coordinates for where the dart would land

- Given an (x,y), how can the computer tell if it landed in the circle
  - Make it simple, use the unit circle, and only throw darts at the upper right quadrant
  - Calculate the distance from 0,0
    - **Just calculate the hypotenuse of the triangle**
  - If *hyp* < 1, then the point falls within the unit circle!



Java Algorithm

```
class Pi
   function main
     get number of threads from the command line
        argument as numThreads
     create four objects of class Monte
       passing numThreads / 4 to each of their constructors
     for each Runnable object
                         create an object of class Thread and pass the Runnable
          to its constructor
                         start the thread object
     end for
     wait for 4 threads
     sum answer from each of the four Monte objects into result
     print result
    end function main
end class main
```

```
class Monte implements Runnable
   has integer numIterations
   has double answer

   function run
                          create random number generator
                          set numInside to 0
                          loop numIterations times
                            set x to new random number
                            set y to new random number
                            calculate hyp = square root of x^2+y^2
                            if hyp < 1.0
          add 1 to numInside
       end if
     end loop
     set answer to numInside / numIterations
   end function run


   function constructor(iters)
      set numIterations to iters
   end function constructor


end class MyRunnable
```

Java Code

```java
import java.lang.*;
import java.lang.Math;
import java.util.Random;
import java.util.concurrent.ThreadLocalRandom;

public class Pi {
    public static void main(String[] iters) {
                int numIter = 0;
                if (iters.length < 1) {
                    System.err.println("usage: Pi <iterations>");
                    System.exit(0);
                }
                try {
                    numIter = Integer.parseInt(iters[0]);
                } catch (Exception ex) {
                    System.err.println("Bad argument");
                    System.exit(1);
                }
                Runnable[] runnables = new Runnable[4];
                Thread[] threads = new Thread[4];
                for (int i = 0; i < 4; i++) {
                    runnables[i] = new Monte(numIter/4);
                    threads[i] = new Thread(runnables[i]);
                    threads[i].start();
                }

                double answer = 0;
                try {
                    for (int i = 0; i < 4; i++) {
                                threads[i].join();
                                answer += ((Monte)
runnables[i]).getRatio();
                    }
                } catch (Exception ex) {
                    System.err.println("Thread interrupted");
                    System.exit(2);
                }

                System.out.println("Ratio is: " + answer);

    }
}
```

```java
class Monte implements Runnable {

    private double ratio;
    private int iters;

    public void run() {
                ratio = findRatio(iters);
    }

    public Monte(int iterations) {
                iters = iterations;
    }

    public double getRatio() {
                return ratio;
    }

    private double findRatio(int iterations) {
                ThreadLocalRandom rand = ThreadLocalRandom.current();
                int numIn = 0;
                int numOut = 0;
                for (int i = 0; i < iterations; i++) {
                    // get random number from 0 to 1
                    double x = rand.nextDouble();
                    double y = rand.nextDouble();
                    double hyp = Math.sqrt(x*x + y*y);
                    if (hyp < 1.0) {
                                numIn++;
                    } else {
                                numOut++;
                    }
                }
                return ((numIn + 0.0) / (numIn+numOut));
     }

}
```

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

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

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);
```

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-Local Storage

- **Thread-local storage** (**TLS**) allows each thread to have its own copy of data

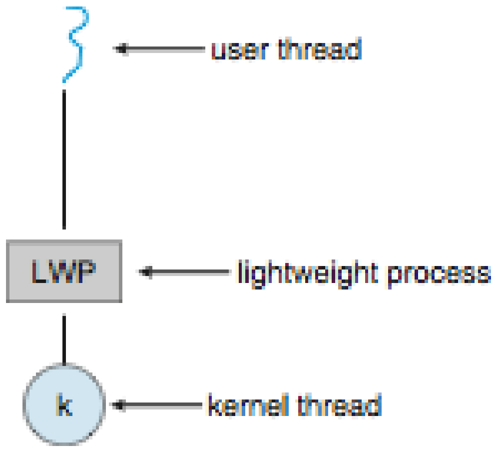
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

- 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



Operating System Examples

- Windows Threads

- Linux Threads

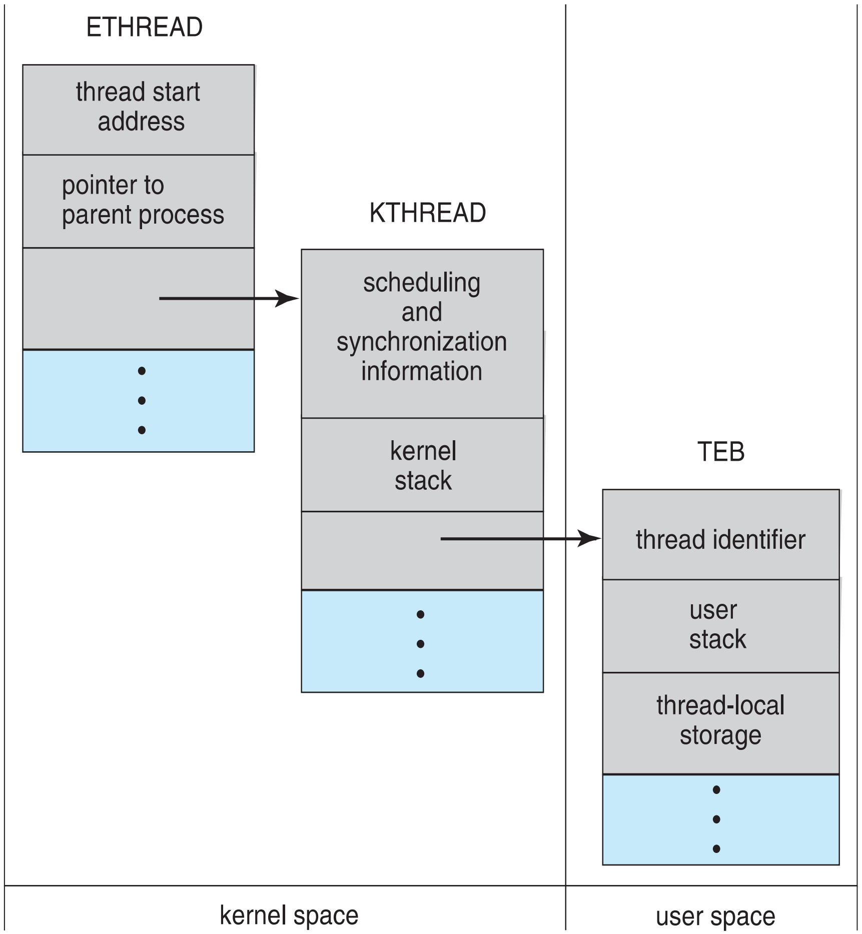
Windows Threads

- Windows implements the Windows API – primary API for Win 98, Win NT, Win 2000, Win XP, and Win 7, 8 and 10
- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - Register set representing state of processor
  - Separate user and kernel stacks for when thread runs in user mode or kernel mode
  - Private data storage area used by run-time libraries and dynamic link libraries (DLLs)
- The register set, stacks, and private storage area are known as the **context** of the thread

Windows Threads (Cont.)

- The primary data structures of a thread include:
  - **ETHREAD** (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
  - **KTHREAD** (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
  - **TEB** (thread environment block) – thread id, user-mode stack, thread-local storage, in user space

Windows Threads Data Structures



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)
  - Flags control behavior

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

- **`struct task_struct`** points to process data structures (shared or unique)