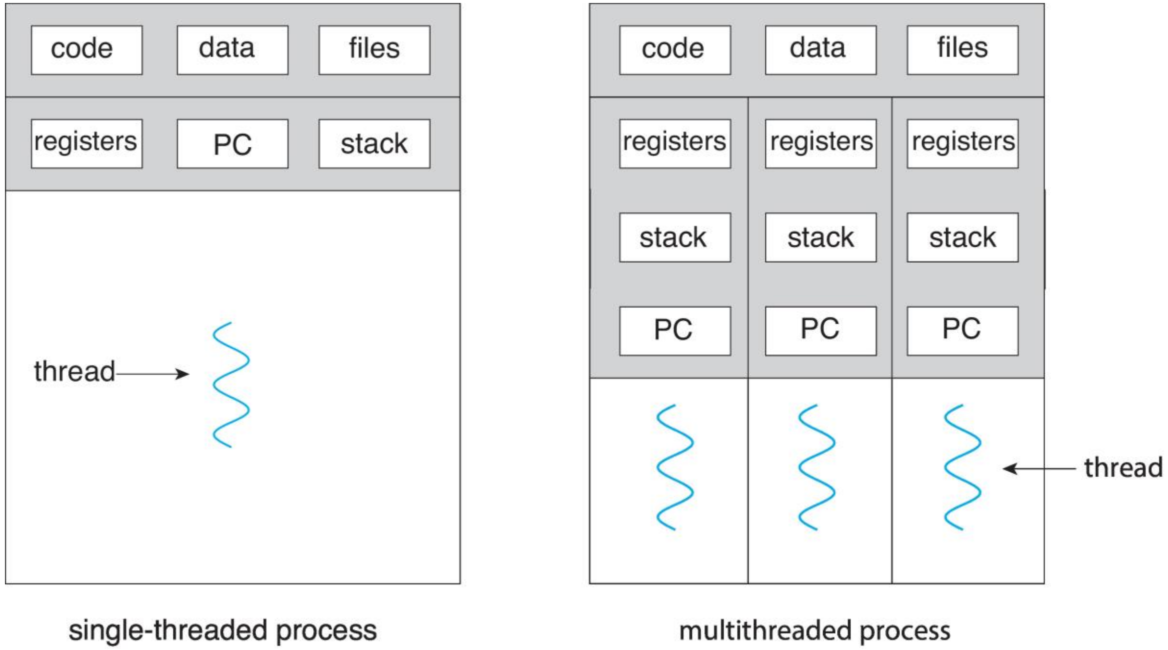
Chapter 4: Threads & Concurrency

Outline

- Process vs. Thread
- Concurrency vs. Parallelism
- Thread Libraries
- Implicit Threading
- Threading Issues
- Example: Windows Threads

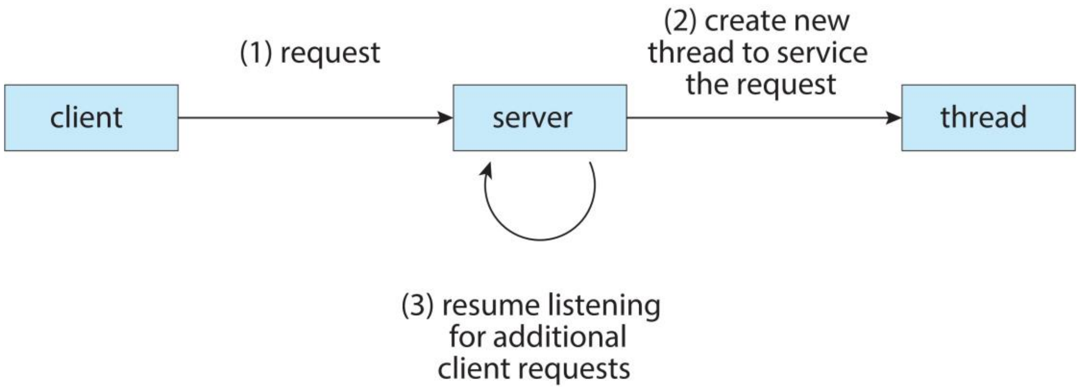
What is a thread

- A thread is a single sequence stream within a process.
- The threads of a process share its executable **code** and the values of its **variables** (code section, data section, OS resources) at any given time.



What is a thread

- Most modern applications are multi-threaded
  - Kernels are generally multi-threaded
- Multiple tasks in an application can be implemented by separate threads, e.g., the following tasks in an application:
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- An example



Motivation (Benefits) Behind Threads

- Economy
  - Process creation： heavy-weight
  - thread creation： light-weight
- Resource Sharing
  - Threads run within the application/process
- Efficiency
  - Can simplify code
- Responsiveness
  - may allow continued execution if part of process is blocked, especially important for user interfaces
- Scalability
  - process can take advantage of multicore architectures, with one or two threads per core

Disadvantages Behind Threads

- More difficult to program with threads (a single process can now do multiple things at the same time).
- New categories of bug are possible (**synchronization** is then required between threads: Chapter 6).

Threads vs. Processes

- Similarities (following attributes own by processes too)
  - Threads share CPU and only one thread active (running) at a time.
  - Threads within a processes execute sequentially.
  - Thread can create children.
  - If one thread is blocked, another thread can run.
- Differences
  - A thread is a component of a process
  - Multiple threads can exist within one process.
  - Multiple threads execute **concurrently** and share resources such as memory, while different processes do not share these resources.

Threads vs. processes Cont.

- More differences

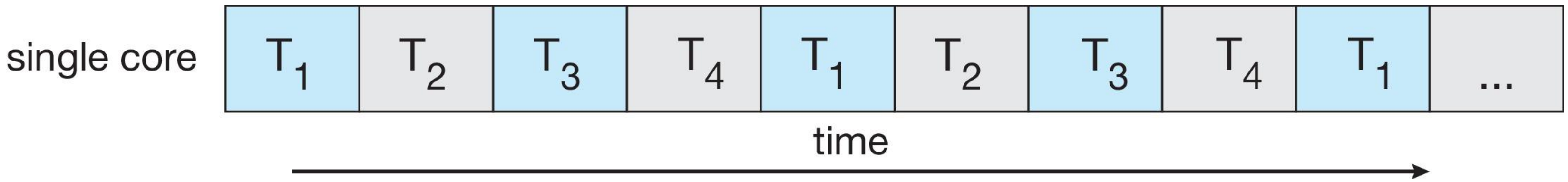
processes	threads
typically independent	subsets of a process
more state information	share process state and resources
separate address spaces	same address space
interact through IPC models: (shared memory/message passing)	variables
slower context switching	faster context switching
might or might not assist one another	designed to assist one another

Multicore Programming

- Multi-core or multi-processor systems putting pressure on programmers, challenges include:
  - Dividing activities
  - Load Balance
  - Data splitting
  - Data dependency
  - Testing and debugging
- Parallelism
  - A system can perform more than one task simultaneously
  - Multiple processors / cores are required
- Concurrency
  - More than one task are progressing
  - Single processor / core, CPU scheduler providing concurrency by doing context switches
- Parallelism implies concurrency, but concurrency does not imply parallelism.

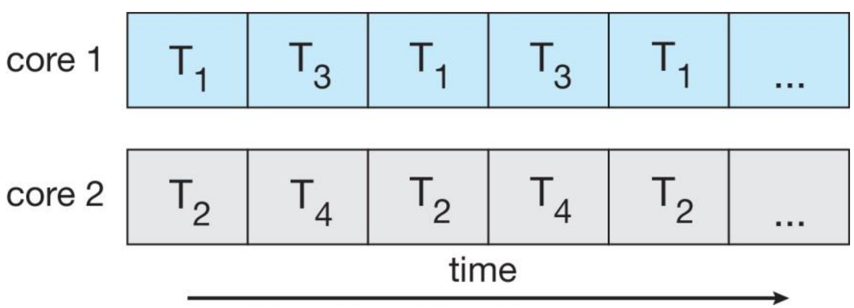
Concurrency vs. Parallelism

- Concurrency is a property of a program where two or more tasks can be in **progress** simultaneously.



Concurrent execution on single-core system

- Parallelism is a run-time property where two or more tasks are being **executed** simultaneously.

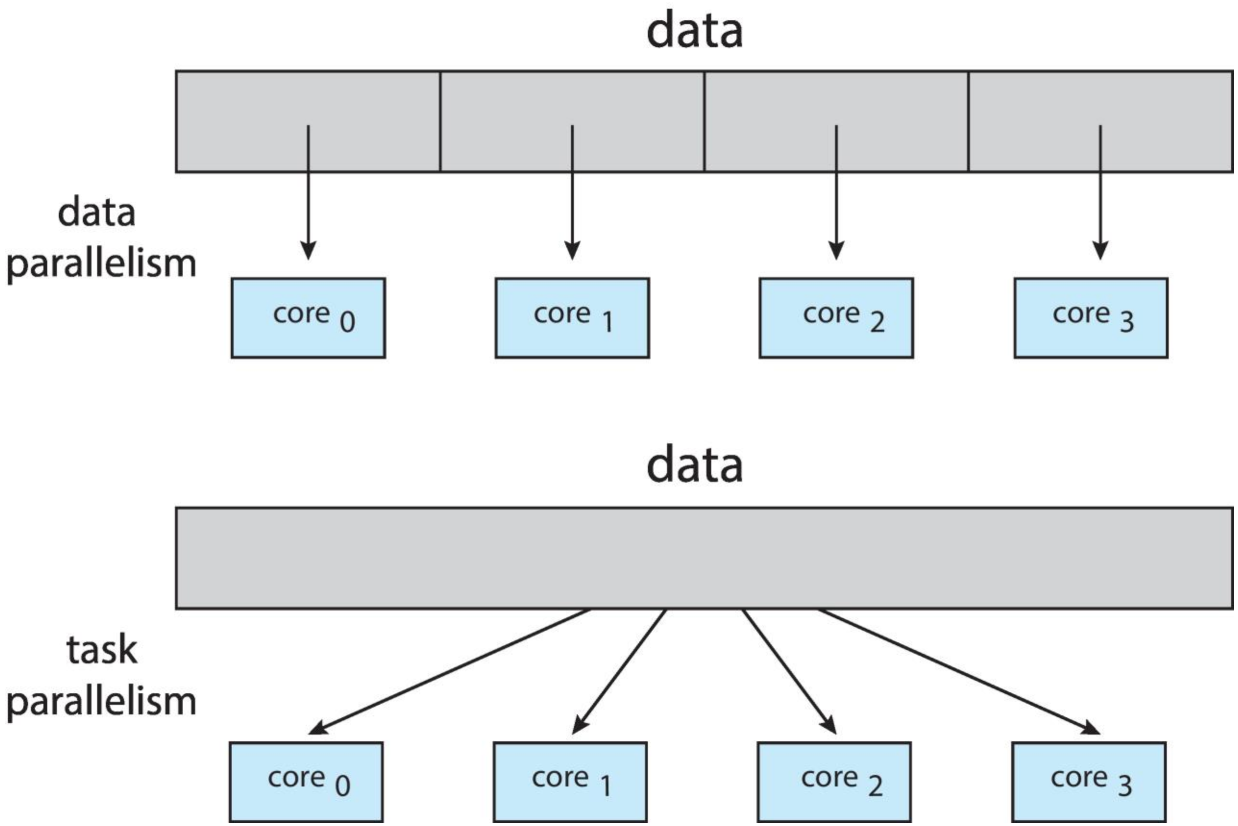


Parallelism on a multi-core system

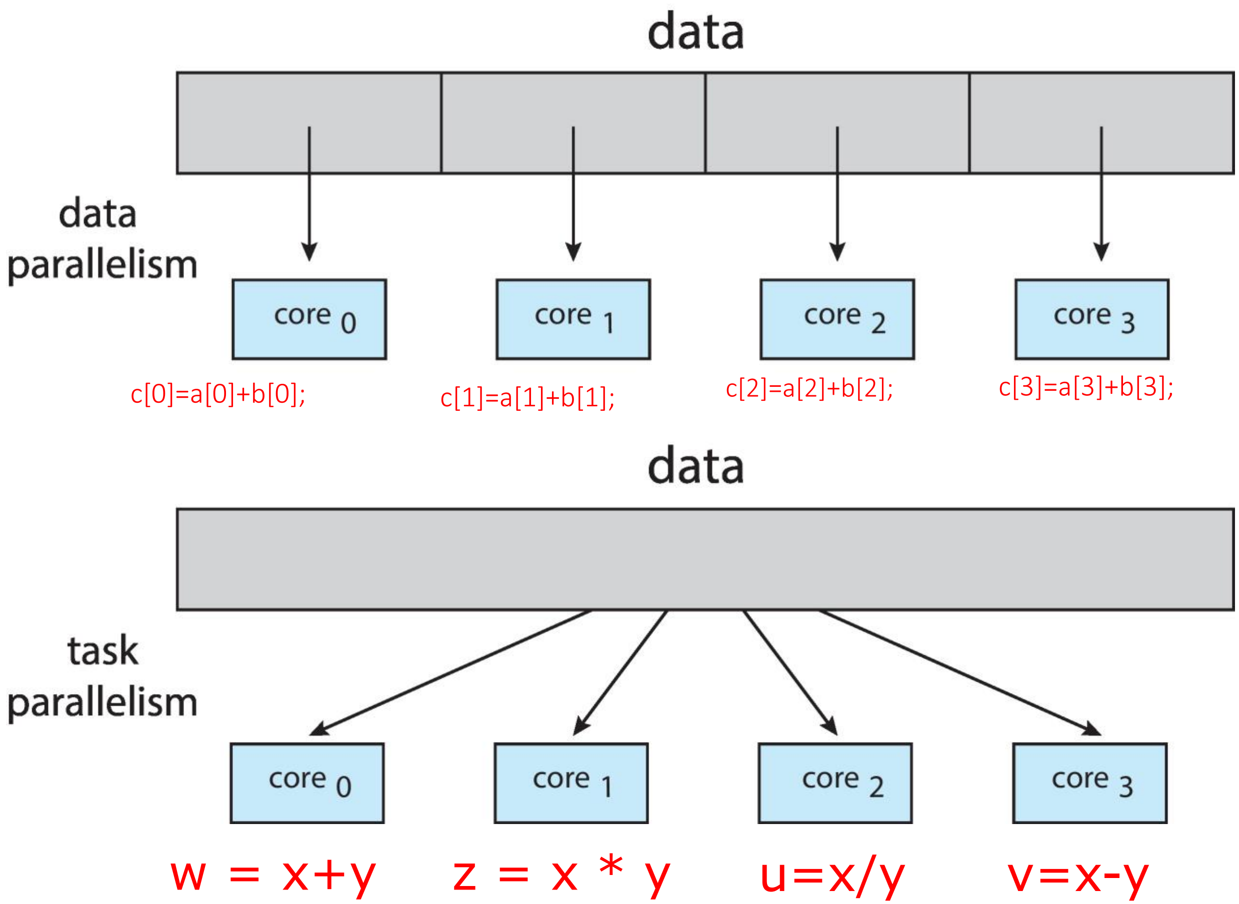
Multicore Programming

- Types of parallelism
  - Data parallelism – distributes subsets of the same data across multiple cores, same operation on each
    - Example: when doing image processing, two cores can each process half of the image
  - Task parallelism – distributing threads across cores, each thread performing unique operation
    - Example: when doing sound processing, the sound data can move through each core in sequence, with each core doing a different kind of sound processing (filtering, echo, etc.)

Data and Task Parallelism



Data and Task Parallelism




```
int a[4], b[4], c[4];
.....
c[0]=a[0]+b[0];
c[1]=a[1]+b[1];
c[2]=a[2]+b[2];
c[3]=a[3]+b[3];
```

```
int x, y, u, v, w, z;
....
w = x+y;
z = x * y;
u=x/y;
v=x-y
```

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

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

Proof:

S is serial portion, P is parallel portion of program.

So $S + P = 100\% = 1$

Assume that running time on one core: $R_1 = S + P = 1$

Then running time on N cores: $R_N \geq S + P/N = S + (1 - S)/N$

(≥, not =, because of extra communication required between threads.)

Therefore, speedup $= R_1 / R_N \leq 1 / (S + (1 - S)/N)$

□

https://en.wikipedia.org/wiki/Amdahl%27s_law

Amdahl's Law Example

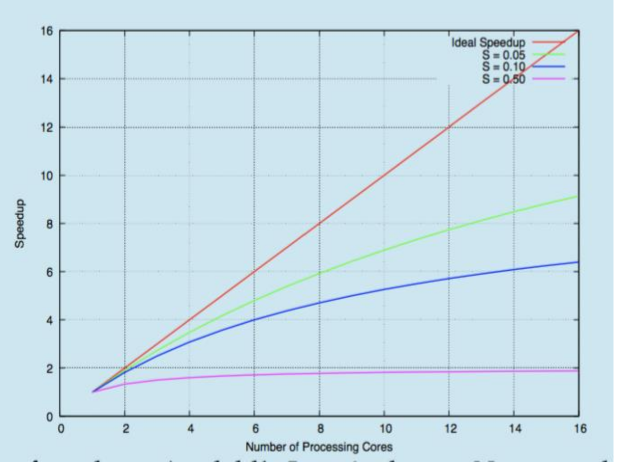
- Example: if the application is 75% parallel and 25% serial, moving from 1 to 2 cores:

  $$0.25 + 0.75/2 = 0.625$$

  results in a maximum speedup of 1/0.625 = 1.6 times.

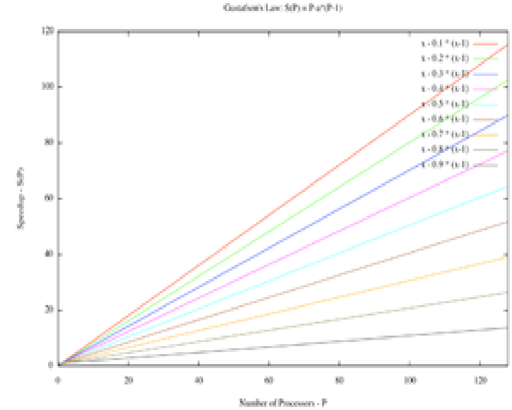
- As *N* approaches infinity, the maximum speedup approaches 1 / *S*
  - Serial portion of an application has disproportionate effect on performance gained by adding additional cores.

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



Comparison : Gustafson's Law

- Gustafson's law addresses the shortcomings of Amdahl's law
- It is based on the assumption of a fixed problem size
  - an execution workload that does not change with respect to the improvement of the resources



$$\textit{speedup} = S + P \times N$$
$$= S + (1 - S) \times N$$
$$= N + (1 - N) \times S$$

where

speedup: theoretical scaled speedup of the program with parallelism.

N, S, P: meanings are same as in Amdahl algorithm.

N: is the number of processors;

S: the fractions of time spent executing the serial parts

P: the fractions of time spent executing the parallel parts

S+P=1

https://en.wikipedia.org/wiki/Gustafson's_law

User Threads and Kernel Threads

- Threads
  - User threads
    - management (thread creation, thread scheduling, etc.) done by user-level threads library.
  - Kernel threads
    - management (thread creation, thread scheduling, etc.) supported by the kernel

User Threads

- Advantages:
  - No need for OS support
  - Works even on very old or very simple OS that does not have system calls for thread management.
  - No system call required
  - Fast: only need a library function call.
- Disadvantage:
  - A process with only one thread gets as much CPU time as a process with many threads.
  - All the thread scheduling inside a process must be done at user level (not done by kernel)
    - Each thread must be nice and cooperate with the other threads in the process and regularly give CPU time to the other threads.
    - Program more complicated to write.

Kernel Threads

- Advantages:
  - Kernel knows how many threads each process contains so it can give more CPU time to the processes with many threads.
  - No need for threads to cooperate for scheduling
    - thread scheduling done automatically by kernel
    - user program simpler to write.
- Disadvantages:
  - Every thread management operation requires a system call
  - Slower compared to user-level threads.
  - Kernel’s PCB data structures more complex
    - the kernel needs to keep track of both processes and threads inside processes.

User Threads and Kernel Threads

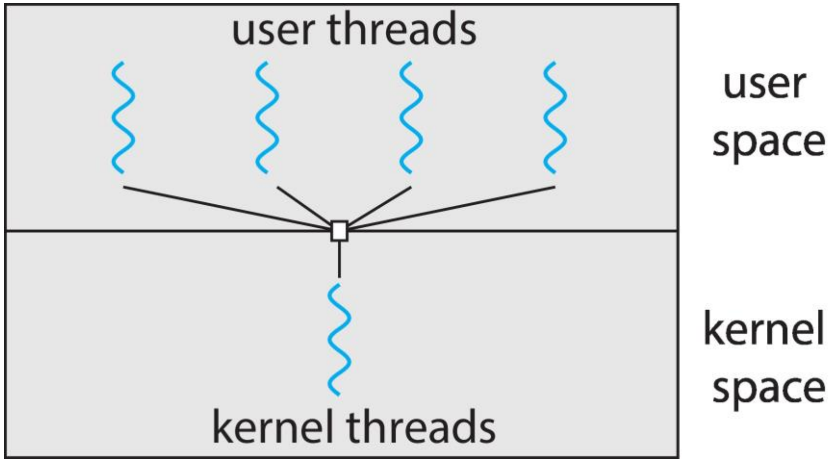
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Linux
  - Mac OS X
  - iOS
  - Android

Multithreading Models

- If threads are available both at user level and kernel level, then some user threads are normally associated with some kernel threads.
- Several models of association between user threads and kernel threads are possible:
  - Many-to-One
  - One-to-One
  - Many-to-Many

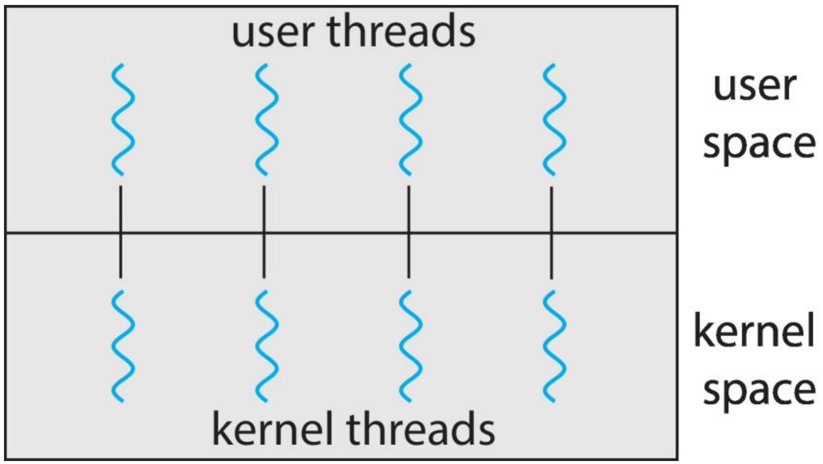
Many-to-One

- Many user-level threads mapped to single kernel thread.
- One thread blocking (waiting for something) causes all threads to block (because their common kernel thread is blocked).
- 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



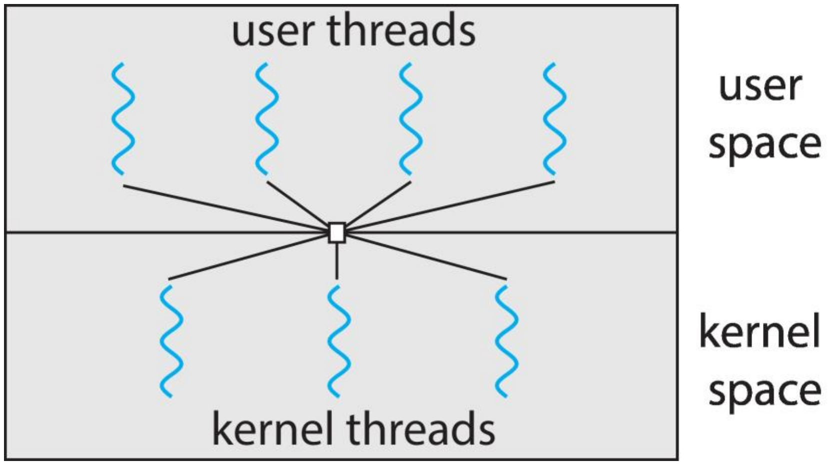
One-to-One

- 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
  - 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.
- Example: Windows with the *ThreadFiber* package.
- Otherwise not very common.



Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementation
  - Library entirely in user space (user threads only)
  - OS-level library supported by the kernel (user threads mapped to kernel threads, with one-to-one model for example).
- Three primary thread libraries:
  1. POSIX Pthreads
  2. Windows threads
  3. Java threads

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 developers of the library
- Common in UNIX operating systems (Linux & Mac OS X)

Pthreads Example 1

gcc –o thrd-posix thrd-posix.c –lpthread
./thrd-posix

thrd-posix.c

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

int main(int argc, char *argv[]) {
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of attributes for the thread */
    int n;
    if (argc != 2) {
        fprintf(stderr,"usage: thrd-posix <integer value>\n");
        return -1;
    }
    n = atoi(argv[1]);
    if (n < 0) {
        fprintf(stderr,"Argument %d must be non-negative\n",n);
        return -1;
    }
    pthread_attr_init(&attr); /* get the default attributes */
    pthread_create(&tid,&attr,runner,argv[1]); /* create the thread */
    pthread_join(tid,NULL); /* now wait for the thread to exit */
    printf("sum = %d\n",sum);
}
```

Pthreads Example 1

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

   sum = 0;

   if (upper > 0) {
   for (i = 1; i <= upper; i++)
      sum += i;
   }

   pthread_exit(0);
}
```

Pthreads Example 2: Code for Joining 4 Threads

gcc –o thrd-demo thrd-demo.c –lpthread
./thrd-demo

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 4
void* threadfunc(void *r) {
		printf("This is a pthread %d.\n", *(int*)r);
		pthread_exit(0);
}
int main(void) {
	pthread_t workers[NUM_THREADS];
	int i, ret;
	int data[NUM_THREAD] = {1,2,3,4};

	for (i = 0; i < NUM_THREADS; ++i){
		ret = pthread_create(&workers[i], NULL, threadfunc,
		(void*)&data[i]);
		if (ret != 0){
				printf("Create pthread %d error!\n“, i);
				return 1;
		}
	}
	printf("This is the main process.\n");
	for (i = 0; i < NUM_THREADS; ++i)
		pthread_join(workers[i], NULL);
	return 0;
}
```

thrd-demo.c

More Examples

- More examples on Windows and Java refer to the appendix part in this lecture note

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
- Five methods explored
    1. Thread Pools
    2. Fork-Join
    3. OpenMP (http://www.openmp.org/)
    4. Grand Central Dispatch[1]
    5. Intel Threading Building Blocks (TBB)[2]

This lecture introduces only first three methods briefly

[1]a technology developed by Apple Inc. to optimize application support for systems with multi-core processors and other **symmetric multiprocessing** systems. It is an implementation of task parallelism based on the **thread pool pattern**.

[2]Threading **Building Blocks** (**TBB**) is a C++ template library developed by Intel for parallel programming on multi-core processors. Using TBB, a computation is broken down into tasks that can run in parallel. The library manages and schedules threads to execute these tasks.

Thread Pools

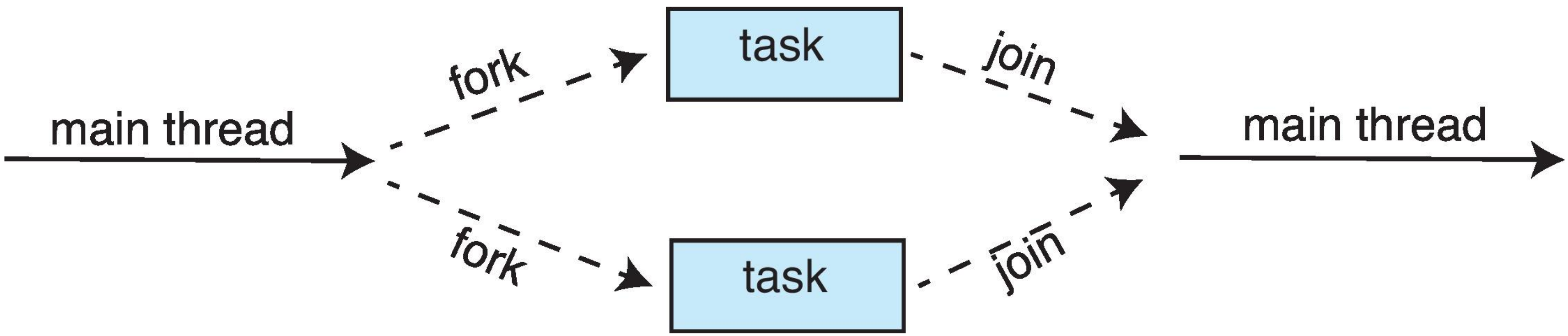
- Create a number of threads in advance in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
  - Separating task to be 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.
   */
}
```

- Available in Java as well.

Fork-Join Parallelism

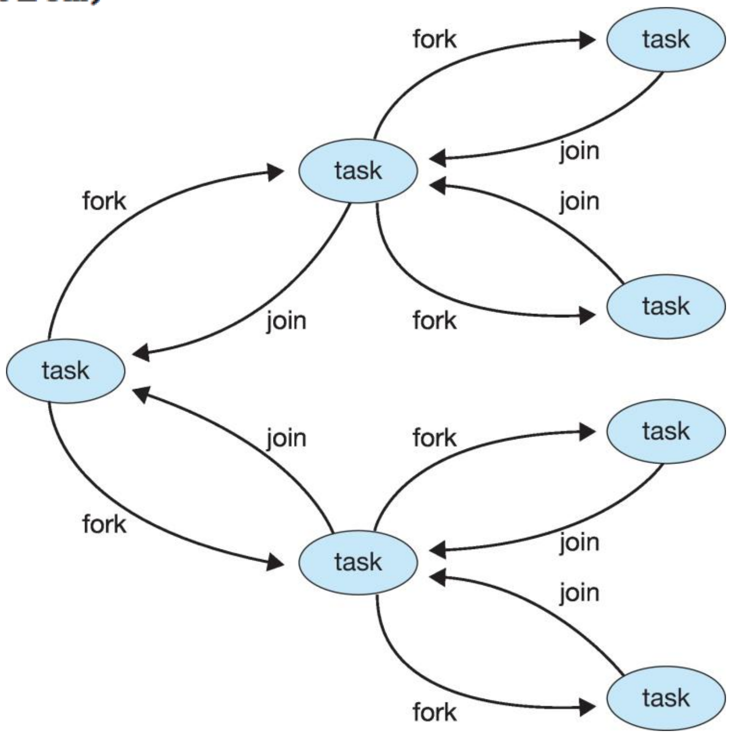
- Multiple threads (tasks) are forked, and then joined.
  - Available in Java.（since Java SE7)
  - Similar to Hadoop MapReduce operation



Fork-Join Parallelism

- General algorithm for fork-join strategy:

```
Task(problem)
   if problem is small enough
      solve the problem directly
   else
      subtask1 = fork(new Task(subset of problem)
      subtask2 = fork(new Task(subset of problem)

      result1 = join(subtask1)
      result2 = join(subtask2)

      return combined results
```



Fork-join calculation Code Example (Java)

```
import java.util.concurrent.*;
public class SumTask extends RecursiveTask<Integer> {
     private static final long serialVersionUID = 1L;
     static final int THRESHOLD = 100;
     private int begin;
     private int end;
     private int[] array;
     public SumTask(int begin, int end, int[] array) {
          this.begin = begin;
          this.end = end;
          this.array = array;
     }
     protected Integer compute() {
          if (end - begin < THRESHOLD) {
               int sum = 0;
               for (int i = begin; i <= end; i++)
                    sum += array[i];
               return sum;
          }
          else {
               int mid = (begin + end) / 2;
               SumTask leftTask = new SumTask(begin, mid, array);
               SumTask rightTask = new SumTask(mid + 1, end, array);
               leftTask.fork();
               rightTask.fork();
               return rightTask.join() + leftTask.join();
          }
     }
}
```

Fork-join calculation Code Example

```
import java.util.concurrent.*;
public class SumDemo {
     private static final int MAX = 1000;

     // creates a random array of integers
     public static int[] createRandomArray(int n) {
          java.util.Random r = new java.util.Random();
          int[] numbers = new int[n];
          for (int i = 0; i < MAX; i++)
               numbers[i] = Math.abs(r.nextInt()%10);
          return numbers;
     }

     public static void main(String[] args) {
          int[] numbers = createRandomArray(MAX); // create the random array

          // display the array
          for (int i = 0; i < numbers.length; i++)
               System.out.println(numbers[i]);

          SumTask rootTask = new SumTask(0, numbers.length-1, numbers);
          ForkJoinPool pool = new ForkJoinPool();
          pool.invoke(rootTask);

          System.out.println(rootTask.compute());
     }
}
```

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

Example: run the for loop in parallel:

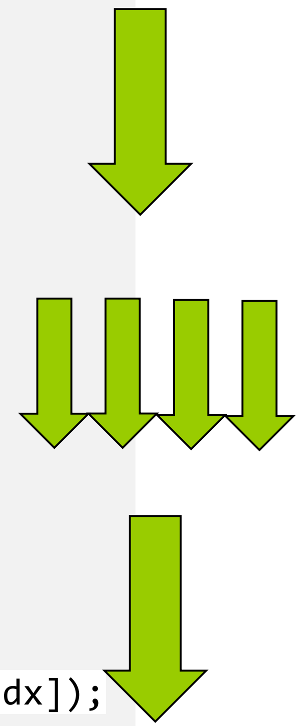
```
#pragma omp parallel for
for (i = 0; i < N; i++) {
   c[i] = a[i] + b[i];
}
```

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

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

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

  /* sequential code */

  return 0;
}
```

OpenMP

openMP_demo.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(int argc, char *argv[]) {
     int a[1000], b[1000], c[1000];
     /* sequential code */
     srand((unsigned)time(NULL));
     for (int i = 0; i < 1000; i++) {
          a[i] = rand() * 100 /RAND_MAX;
          b[i] = rand() * 100 / RAND_MAX;
     }
     /* parallel code */
     #pragma omp parallel for
     for (int i = 0; i < 1000; i++) {
          c[i] = a[i] + b[i];
     }
     /* sequential code */
     for (int i = 0; i < 100; i++) {
          for (int j = 0; j < 10; j++) {
               int idx = 10 * i + j;
               printf("%d+%d=%d\n", a[idx], b[idx], c[idx]);
          }
     }
     return 0;
}
```



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

Linux declare a TLS variable:

_ _thread int number;

Thread-Local Storage

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

__thread int var = 5;

void* worker1(void* arg);
void* worker2(void* arg);

int main(){
    pthread_t pid1,pid2;

    pthread_create(&pid1,NULL,worker1,NULL);
    pthread_create(&pid2,NULL,worker2,NULL);

    pthread_join(pid1,NULL);
    pthread_join(pid2,NULL);

    return 0;
}

void* worker1(void* arg){
            var++;
            printf("work1: %d\n",var);
}

void* worker2(void* arg){
            sleep(1); //sleep for 1s
            var += 2;
            printf("work2: %d\n",var);
}
```

TLC_demo.c

gcc –o TLC_demo TLC_demo.c –lpthread

./TLC_demo

What are the outputs?

Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is target thread
- Two general approaches:
  1. Asynchronous cancellation terminates the target thread immediately
  2. 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

pthread_kill(pthread_t tid, int signal)

Example: Windows Threads

- Windows API – primary API for Windows applications
- 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 data storage area are known as the context of the thread

End of Chapter 4

Appendices

- The appendix parts are for students who are interested in knowing more about the programming related to communications introduced in this lecture.

Windows Multithreaded C Program

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

DWORD sum; // Data shared by all the threads.

// The function executed by the new thread.
DWORD WINAPI runner(LPVOID param) {
    // The new thread is running inside the same process as the main
    // thread, and both threads share the sum variable in memory.
    DWORD Upper = *(DWORD *) param;
    sum = 0;
    for(int i = 0; i <= Upper; i++) {
        sum += i;
        //printf("new thread: sum is: %d\n", sum);
    }
    return 0; // New thread ends.
}
```

Windows Multithreaded C Program (Cont.)

```
int main(int argc, char *argv[]) {
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;
    // do some basic error checking
    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;
    }
    // create the thread
    ThreadHandle = CreateThread(NULL, 0, runner, &Param, 0, &ThreadId);
    if (ThreadHandle != NULL) {
        WaitForSingleObject(ThreadHandle, INFINITE);
        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:
  - Extending Thread class
  - 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) { }
```

Java Multithreaded Program

```
class Sum {
        private int sum;
        public int get() {
                return sum;
        }
        public void set(int sum) {
                this.sum = sum;
        }
}
class Summation implements Runnable {
        private int upper;
        private Sum sumValue;
        public Summation(int upper, Sum sumValue) {
                if (upper < 0)
                        throw new IllegalArgumentException();
                this.upper = upper;
                this.sumValue = sumValue;
        }

        public void run() {
                int sum = 0;
                for (int i = 0; i <= upper; i++)
                        sum += i;
                sumValue.set(sum);
        }
}
```

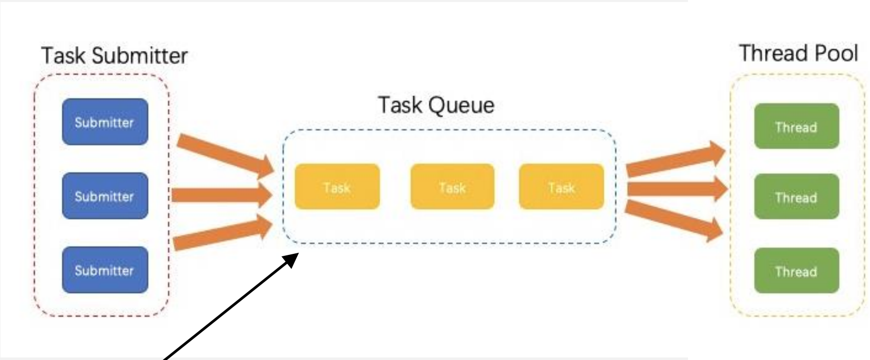
Java Multithreaded Program (Cont.)

```
public class ThreadDemo {
    public static void main(String[] args) {
        if (args.length != 1) {
                System.err.println("Usage ThredDemo <integer>");
                System.exit(0);
        }

        Sum sumObject = new Sum();
        int upper = Integer.parseInt(args[0]);

        Thread worker = new Thread(new Summation(upper, sumObject));
        worker.start();

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

        System.out.println("The sum of " + upper + " is " + sumObject.get());
    }
}
```

Java Multithreaded Program (Cont.)

Since SE5
Thread pool

```
import java.util.concurrent.*;
class Summation implements Callable<Integer> {
        private int upper;
        public Summation(int upper) {
                this.upper = upper;
        }
        /* The thread will execute in this method */
        public Integer call() {
                int sum = 0;
                for (int i = 1; i <= upper; i++)
                        sum += i;
                return new Integer(sum);
        }
}
public class Driver {
        public static void main(String[] args) {
                int upper = Integer.parseInt(args[0]);
                ExecutorService pool = Executors.newSingleThreadExecutor();
                Future<Integer> result = pool.submit(new Summation(upper));
                try {
                        System.out.println("sum = " + result.get());
                }
                catch (InterruptedException | ExecutionException ie) { }
        }
        //shutdown the pool
        pool.shutdown();
}
```



Java Multithreaded Server

```
import java.net.*;
import java.io.*;

public class Connection implements Runnable {
        private Socket          outputLine;

        public Connection(Socket s) {
                outputLine = s;
        }

        public void run() {
                // getOutputStream returns an OutputStream object
                // allowing ordinary file IO over the socket.

                try {
                        // create a new PrintWriter with automatic flushing
                        PrintWriter pout = new PrintWriter(outputLine.getOutputStream(), true);

                        // now send the current date to the client
                        pout.println(new java.util.Date() );

                        // now close the socket
                        outputLine.close();
                }
                catch (java.io.IOException e) {
                        System.out.println(e);
                }
        }
}
```

Java Multithreaded Server (Cont.)

```
public class Server {
        private ServerSocket    s;
        private Socket          client;
        public Server(){
                // create the socket the server will listen to
                try {
                        s = new ServerSocket(6013);
                }
                catch (java.io.IOException e) {
                        System.out.println(e);
                        System.exit(1);
                }
                // OK, now listen for connections
                System.out.println("Server is listening ....");
                try {
                        while (true) {
                                client = s.accept();

                                // create a separate thread
                                // to service the request
                                (new Thread(new Connection(client))).start();
                        }
                }
                catch (java.io.IOException e) {
                        System.out.println(e);
                }
        }
        public static void main(String args[]) {
                Server fortuneServer = new Server();
        }
}
```

Java Multithreaded Server (Cont.)

```
import java.io.*;
import java.net.*;
public class Client {
    private Socket CSocket = null;
    private PrintWriter out = null;
    private BufferedReader in = null;
    public Client() throws IOException {
        try {
            CSocket = new Socket("localhost", 6013);//server IP and port
            out = new PrintWriter(CSocket.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader(CSocket.getInputStream()));
            System.out.println("Socket is created successfully!");
        } catch (IOException e) {
            System.err.println("I/O exception: "+e.getMessage());
            System.exit(1);
        }
    }
    public static void main(String[] args) {
        try{
            Client c = new Client();
            String fromServer = c.in.readLine();
            System.out.println(fromServer);
        }catch(IOException e){
            System.err.println(e.getMessage());
        }
    }
}
```