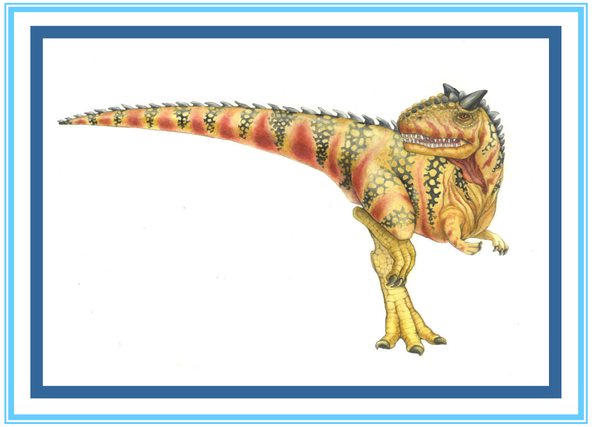
Chapter 4: Multithreaded Programming



Chapter 4: Multithreaded Programming

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

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

- A **thread in computer** science is short for a **thread** of execution. **Threads** are a way for a program to divide (termed "split") itself into two or more simultaneously (or pseudo-simultaneously) running tasks.

- 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

Thread Concept

- Single threaded application

```
int main(){
        ..
        f1();
        printf(`Done\n`);
}

int f1(){
        ......
        return result;
}
```

- Multithreaded application

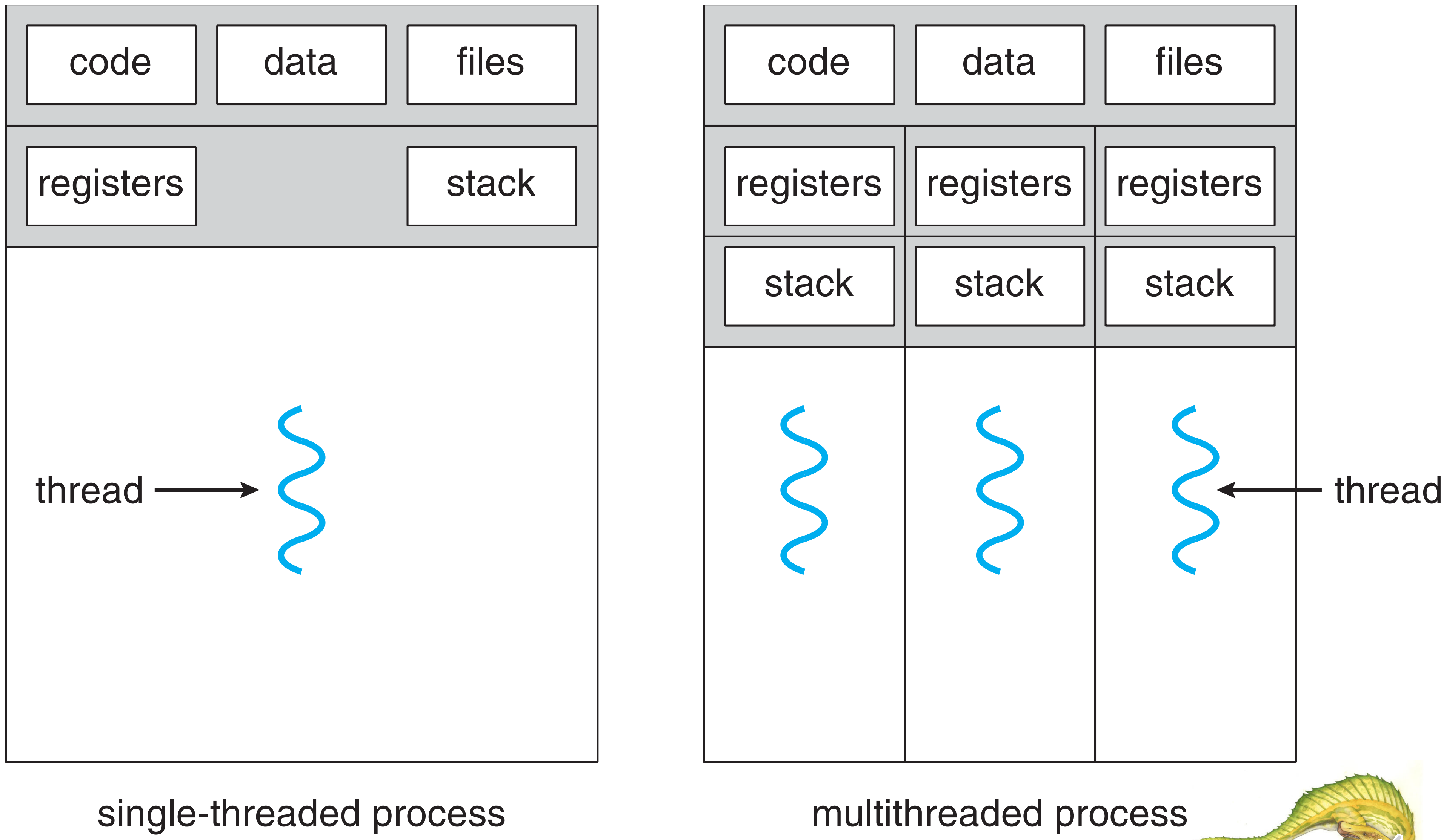
```
int main(){
        ..
        f2();
        printf(`Done\n`);
}

int f2(){
        ......
        return result;
}
```

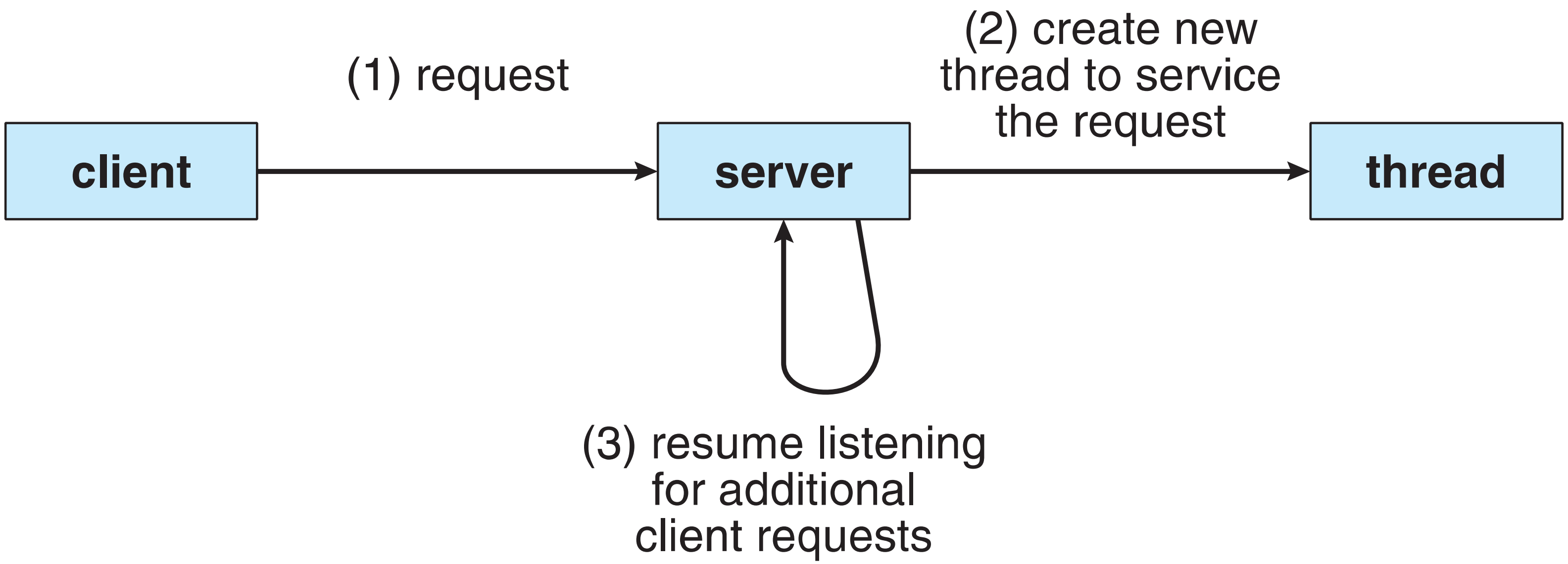
```
int th1(){
        ......
}
```

```
int th2(){
        ......
}
```

Single and Multithreaded Processes



Multithreaded Server Architecture



Benefits

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

Resource Sharing

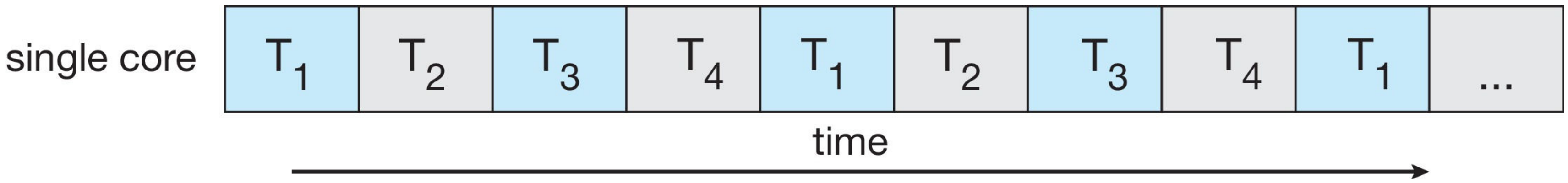
- All of threads of a process share the same memory space and open files.
- Within the shared memory, each thread gets its own stack.
- Each thread has its own instruction pointer and registers.
- OS has to keep track of processes, and stored its per-process information in a data structure called a process control block (PCB).
- A multithread-aware OS also needs to keep track of threads.
- The items that the OS must store that are unique to each thread are:
  - Thread ID
  - Saved registers, stack pointer, instruction pointer
  - Stack (local variables, temporary variables, return addresses)
  - Signal mask
  - Priority (scheduling information)

Multicore Programming

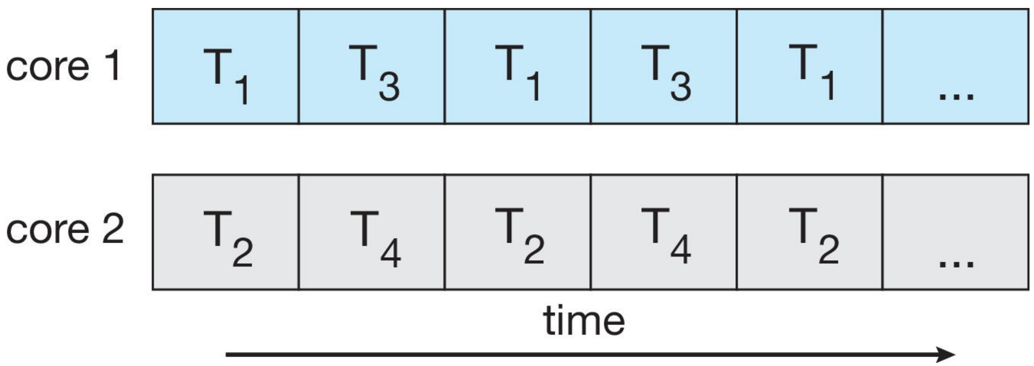
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously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

Concurrency vs. Parallelism

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



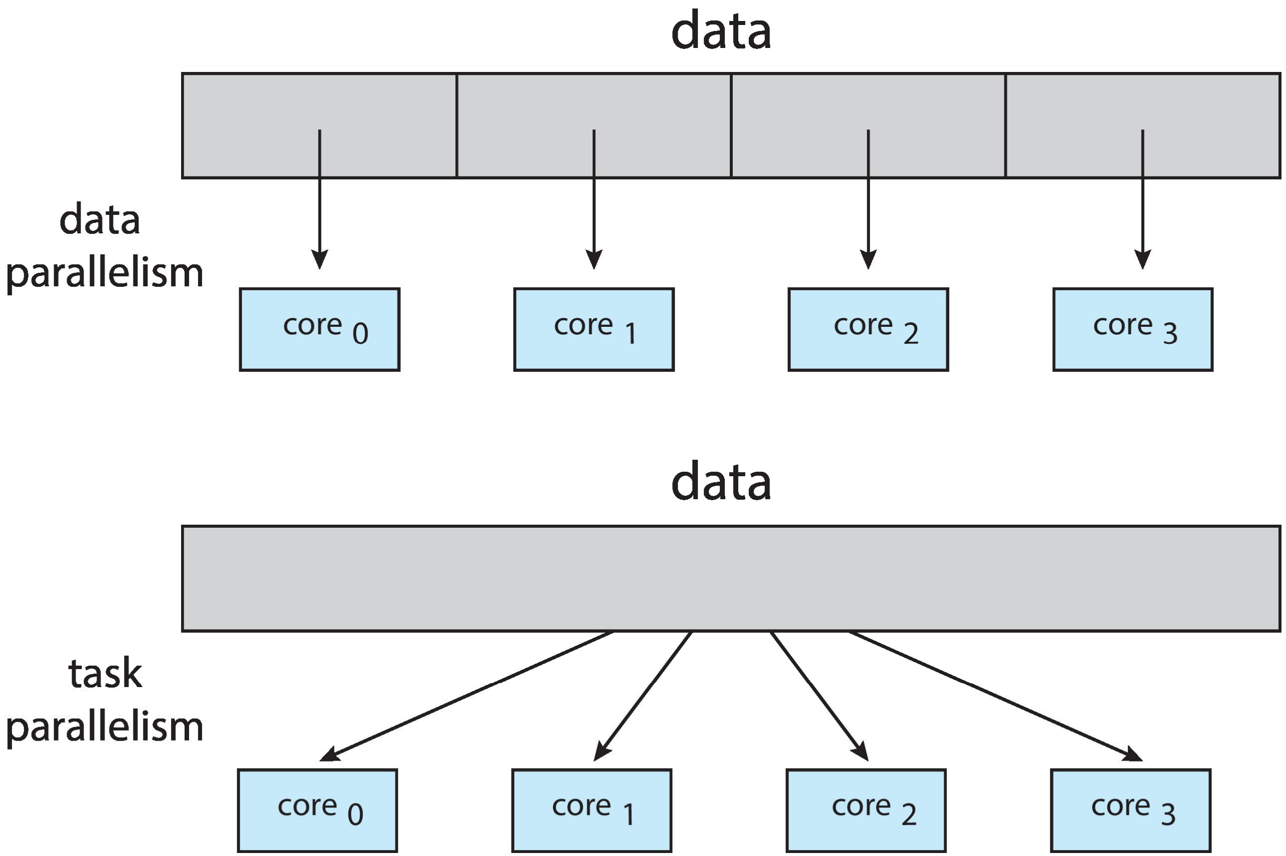
- **Parallelism on a multi-core system:**



Multicore Programming

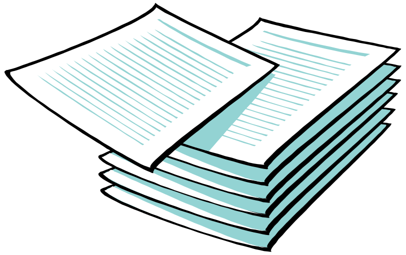
- Types of parallelism
  - **Data parallelism** – distribute subsets of the same data across multiple threads/cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation

Data and Task Parallelism



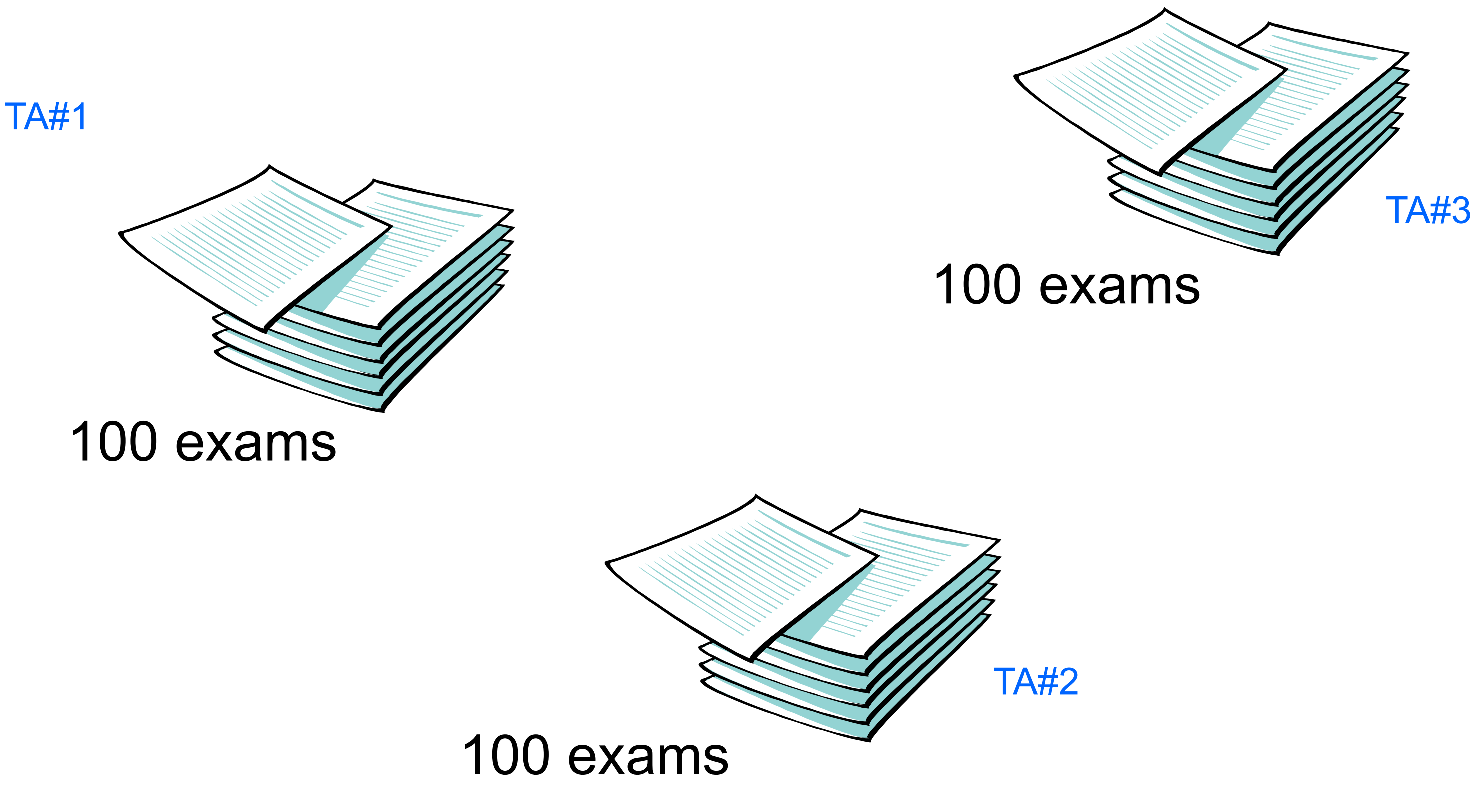
Professor P

15 questions
300 exams



Professor P's grading assistants

Division of work – data parallelism



Division of work – task parallelism

TA#1

Questions 1 - 5

TA#3

Questions 11 - 15

TA#2

Questions 6 - 10

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

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

- I.e. if an application is 25% serial and 75% parallel, moving from 1 to 2 cores results in speedup of
- So, the speedup will be less than $\frac{1}{0.25+\frac{(1-0.25)}{2}} = \frac{1}{0.625} = 1.6$

Example

- Assume that a program's serial execution time is

  $T_{serial} = 20$ seconds

- We can parallelize 90% of the program.
- Parallelization is "perfect" regardless of the number of cores *p* we use.
- Runtime of parallelizable part is

$$0.9 \times T_{serial} / p = 18 / p$$

Example (cont.)

- Runtime of “unparallelizable/serial” part is

$$0.1 \times T_{serial} = 0.1 \times 20 = 2 \text{ seconds}$$

- Overall parallel run-time is

$$T_{parallel} = 0.9 \times T_{serial} / p + 0.1 \times T_{serial} = 18 / p + 2$$

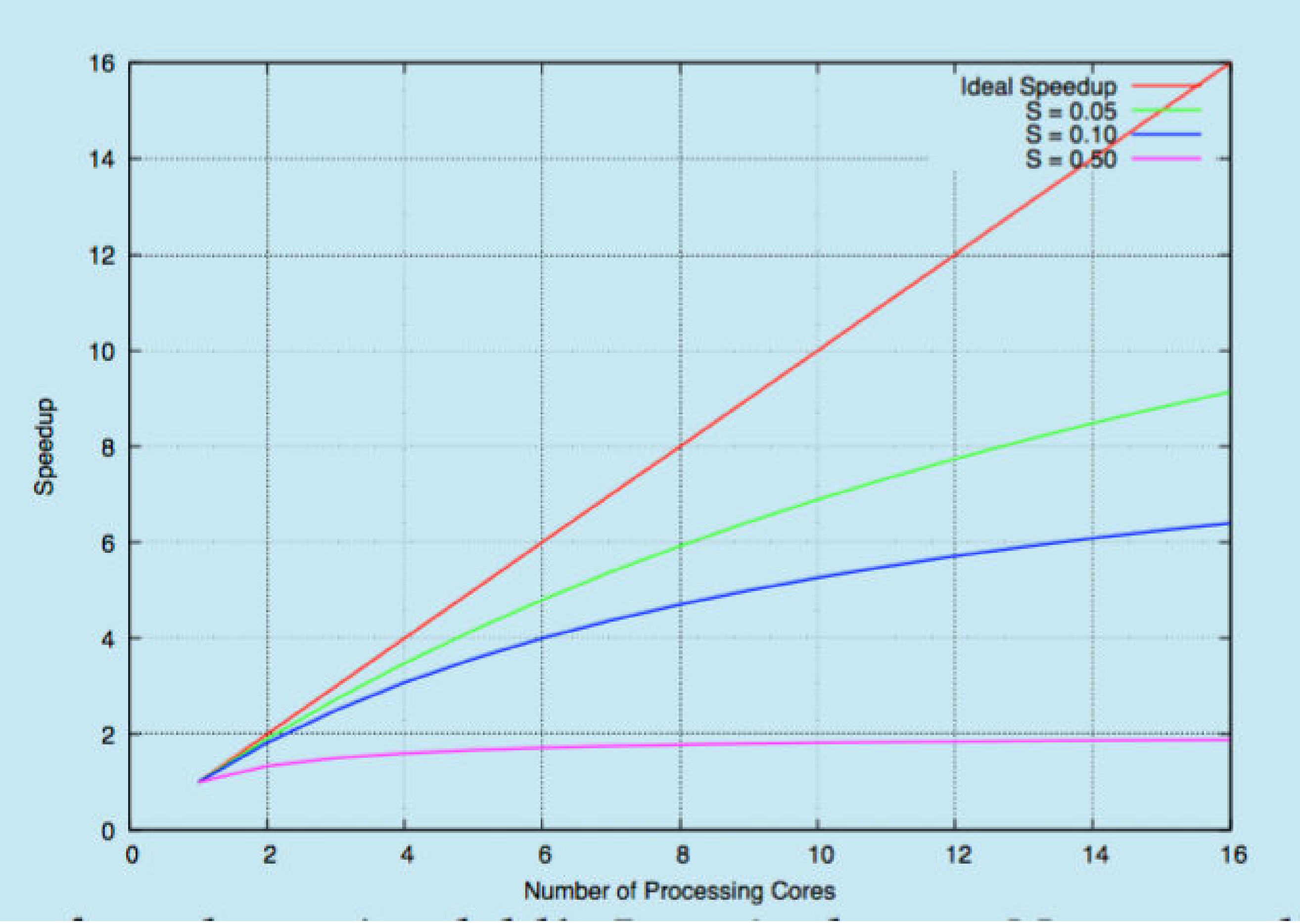
Example (cont.)

- Speed up factor

$$S = \frac{T_{serial}}{0.9 \times T_{serial} / p + 0.1 \times T_{serial}} = \frac{20}{18 / p + 2}$$

If p =10;
20 / 3.8 = ~5.25x speed up

Amdahl's Law

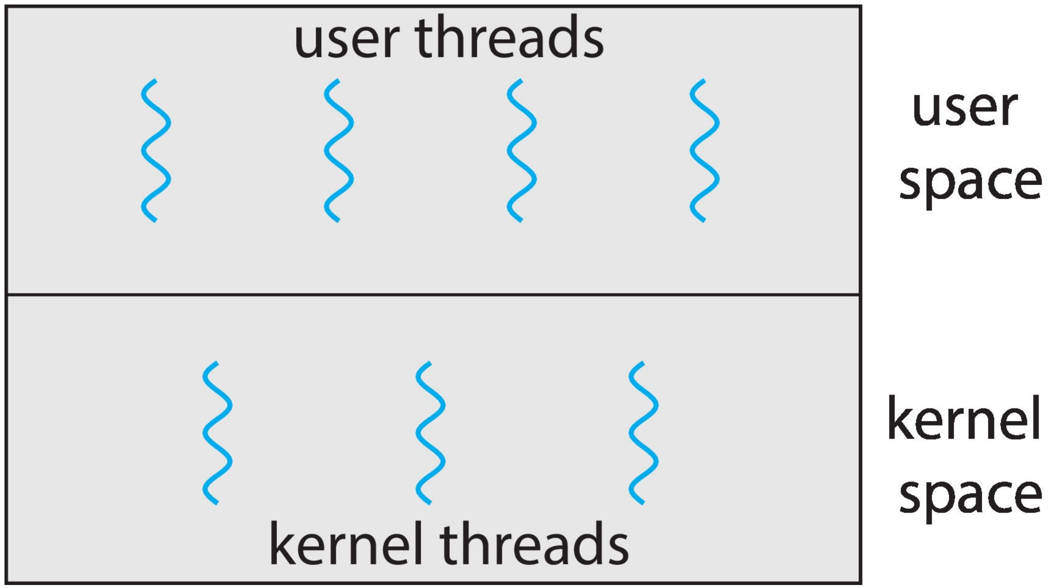


As *N* approaches infinity, speedup approaches 1 / *S*

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

But does the law take into account contemporary multicore systems?

User and Kernel Threads



User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Win32 threads
  - Java threads

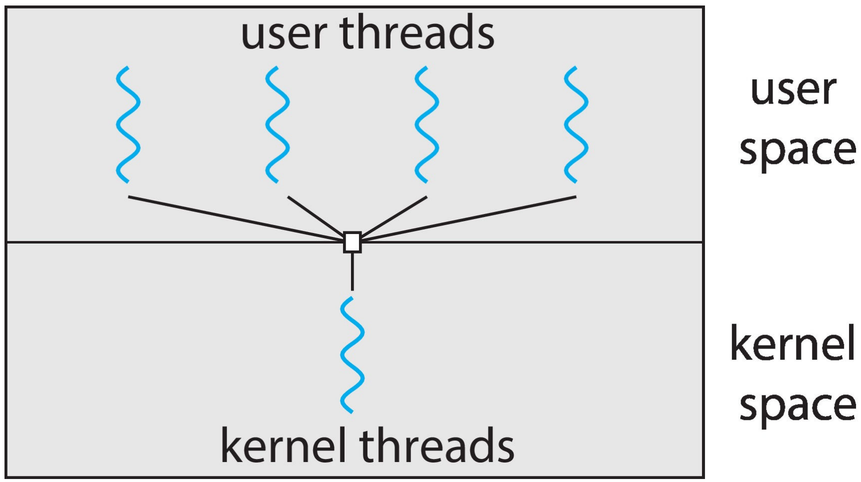
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Multithreading Models

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

Many-to-One

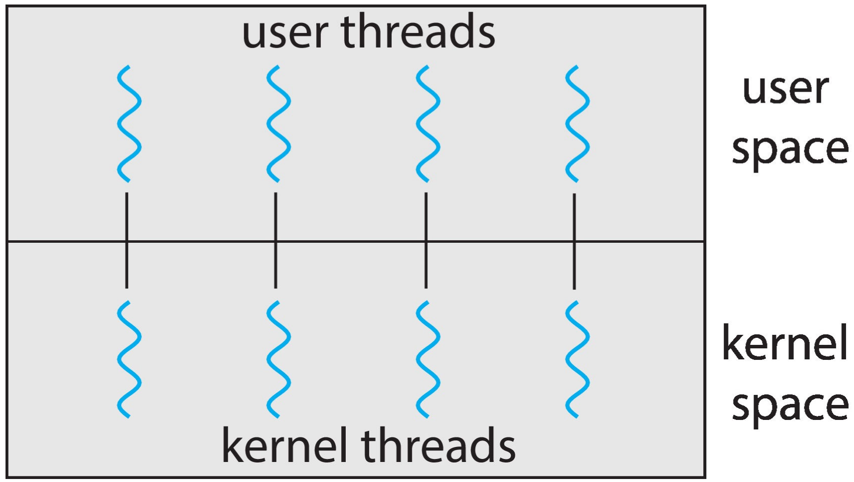
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable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



One-to-One

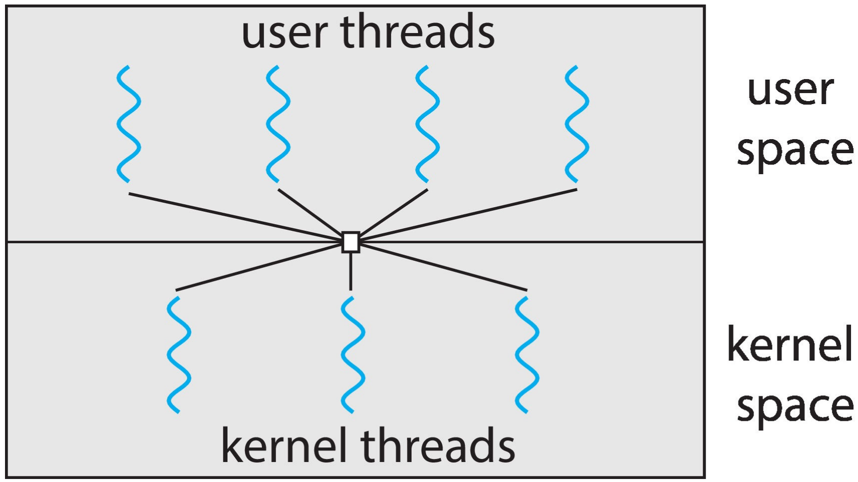
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead

- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later



Many-to-Many Model

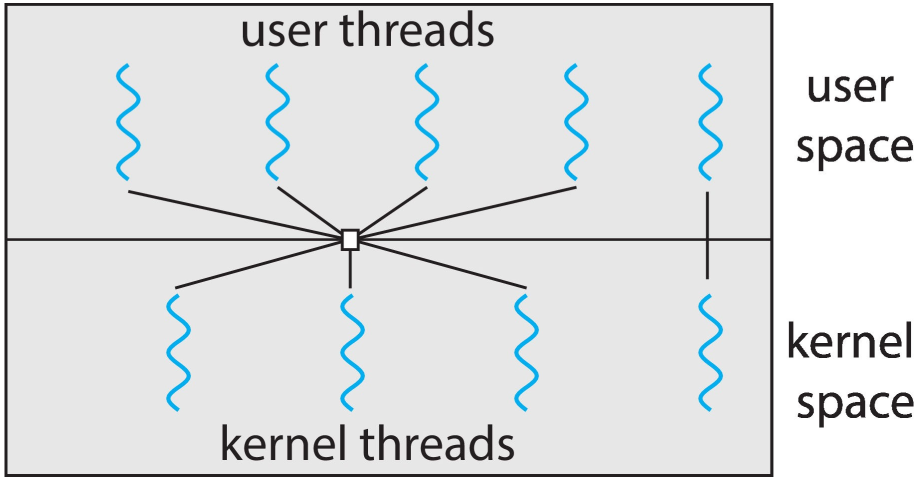
- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads



- Solaris prior to version 9
- Windows NT/2000 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

POSIX threads (Pthreads)

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- May be provided either as user-level or kernel-level
- 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

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread function*/

int main (int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /*attributes for the thread */

   /* get the default attributes */
   pthread_attr_init(&attr);

   /* create the thread*/
   pthread_create(&tid,&attr,runner,argv[1]);

   /* now wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}
```

```c
/* The thread 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 Code for Joining 10 Threads

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */
#define NUM_THREADS 10

int main (int argc, char *argv[])
{
   int i;
   pthread_t workers[NUM_THREADS]; /* the thread array*/
   pthread_attr_t attr; /*attributes for the threads */

   sum = 0;
   /* get the default attributes */
   pthread_attr_init(&attr);

   /* create the thread*/
   for (i=0; i<NUM_THREADS; i++)
         pthread_create(&worker[i], &attr,runner, i+1);

   /* now wait for the thread to exit */
   for (i=0; i<NUM_THREADS; i++)
         pthread_join(worker[i] ,NULL);

   printf("sum = %d\n",sum);
}
```

```c
/* The thread function */

void *runner(void *param)
{
   int i, upper = atoi(param);

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

   pthread_exit(0);
}
```

Windows Multithreaded C Program

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

   Param = atoi(argv[1]);
   /* create the thread */
   ThreadHandle = CreateThread(
      NULL, /* default security attributes */
      0, /* default stack size */
      Summation, /* thread function */
      &Param, /* parameter to thread function */
      0, /* default creation flags */
      &ThreadId); /* returns the thread identifier */

    /* now wait for the thread to finish */
   WaitForSingleObject(ThreadHandle,INFINITE);

   /* close the thread handle */
   CloseHandle(ThreadHandle);

   printf("sum = %d\n",Sum);
}
```

Windows Multithreaded C Program

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

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

Java Threads

- Java threads are managed by the Java Virtual Machine (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();
}
```

Java Multithreaded Program

```
public class Driver
{
  public static void main(String[] args) {

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

     Thread worker = new Thread(new
                        Summation(upper, sumObject));
     worker.start();
     try {
        worker.join();
        System.out.println("The sum of " + upper + " is
           " + sumObject.get());
     } catch (InterruptedException ie) { }
   }
}
```

```
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) {
     this.upper = upper;
     this.sumValue = sumValue;
  }

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

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.
    */
}
```

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;
}
```

Grand Central Dispatch

- Apple technology for Mac OS X and iOS operating systems
- Extensions to C, C++ languages, API, and run-time library
- Allows identification of parallel sections
- Manages most of the details of threading
- Block is in "^{ }" - `^{ printf("I am a block"); }`
- Blocks placed in dispatch queue
  - Assigned to available thread in thread pool when removed from queue
- Two types of dispatch queues:
  - serial – blocks removed in FIFO order, queue is per process, called **main queue**
    - Programmers can create additional serial queues within program
  - concurrent – removed in FIFO order but several may be removed at a time
    - Three system wide queues with priorities low, default, high

```
dispatch_queue_t queue = dispatch_get_global_queue
   (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_async(queue, ^{ printf("I am a block."); });
```

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

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
- 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.)

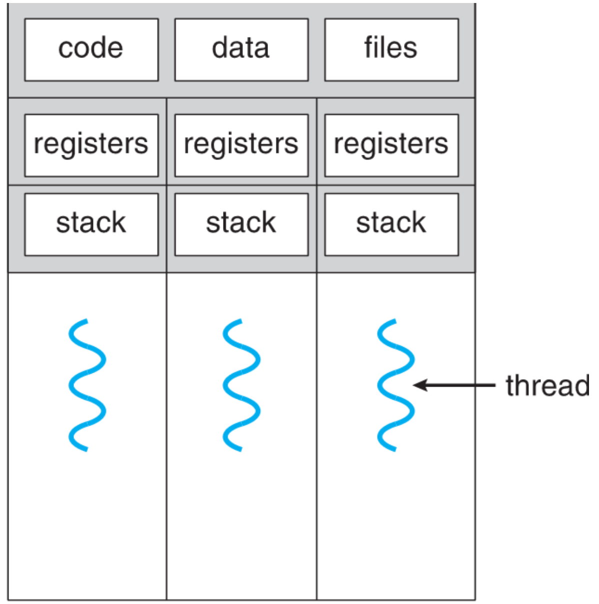
- Invoking thread requests cancellation, but actual cancellation depends on thread state

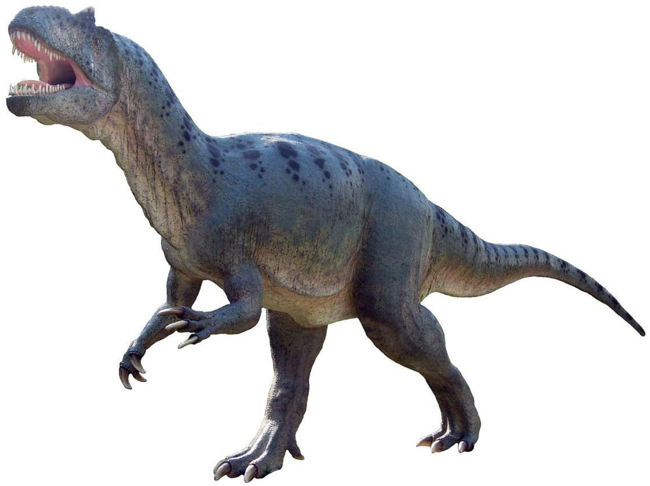
Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

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





Thanks for listening!