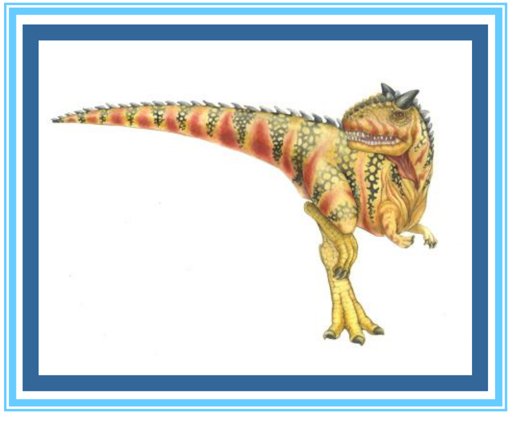
Chapter 4: Threads & Concurrency



Outline

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Operating System Examples

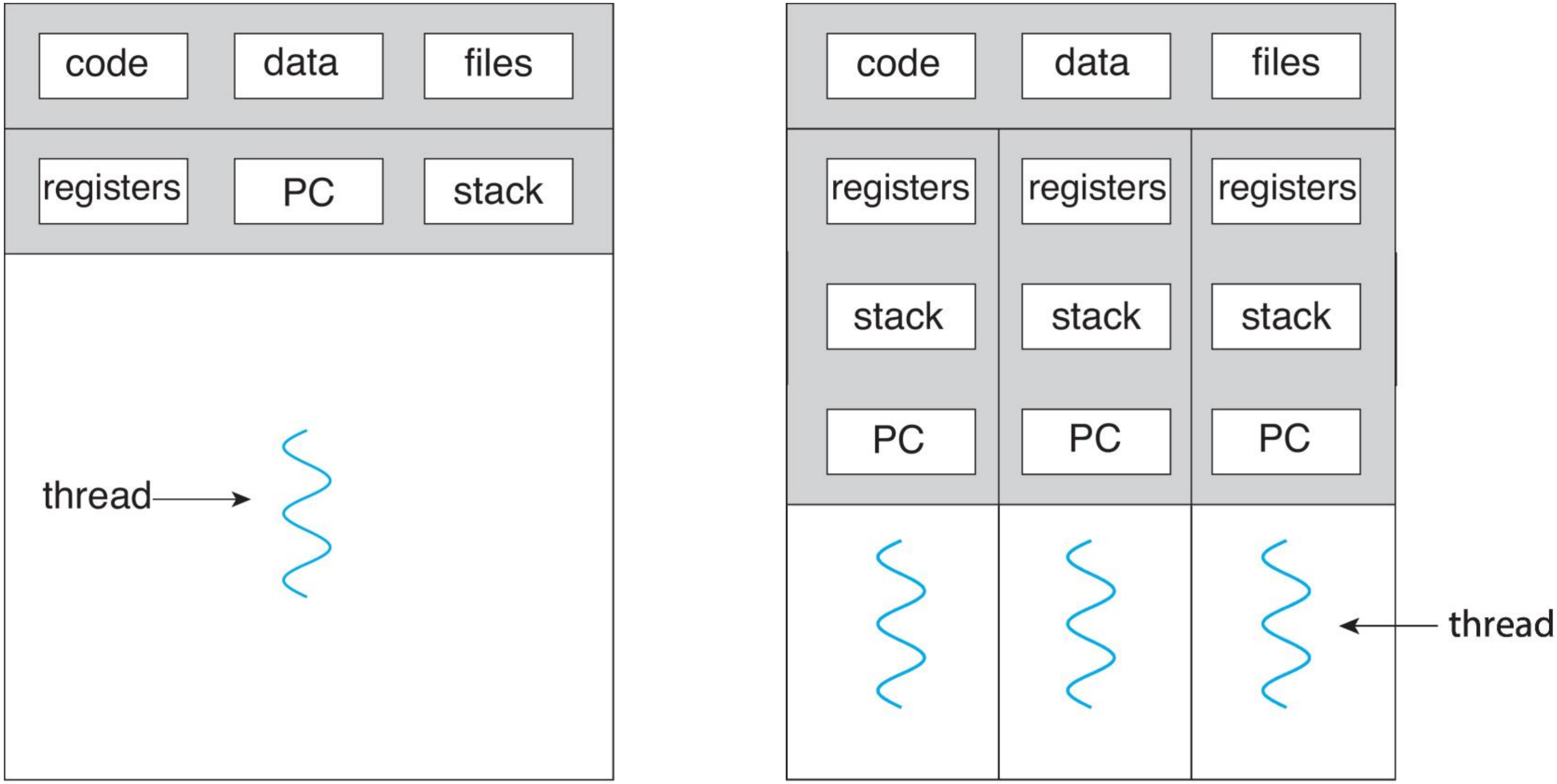
Objectives

- Identify the basic components of a thread, and contrast threads and processes
- Describe the benefits and challenges of designing multithreaded applications
- Designing multithreaded applications using the Pthreads

Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

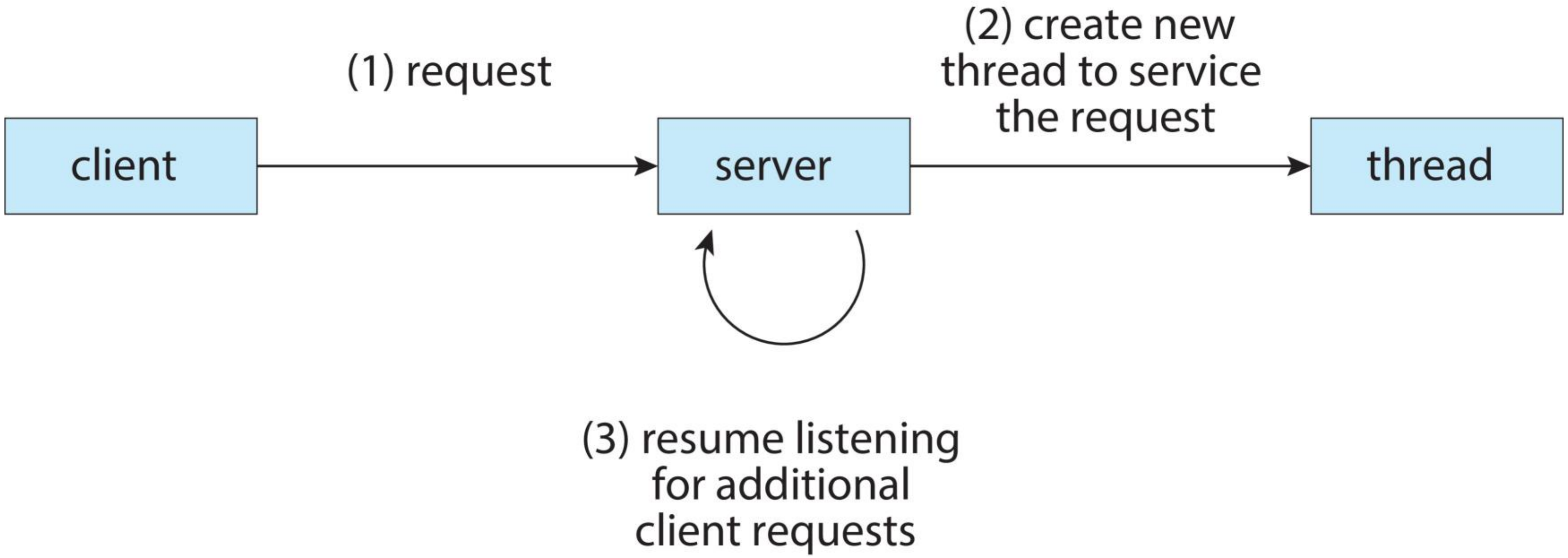
Single and Multithreaded Processes



single-threaded process

multithreaded process

Multithreaded Server Architecture



Benefits

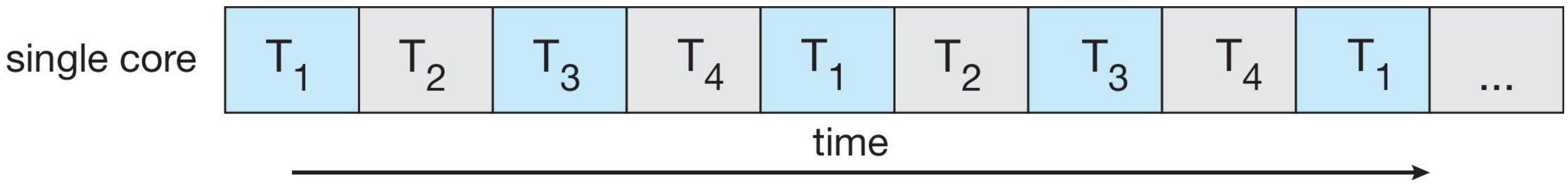
- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower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

Multicore Programming

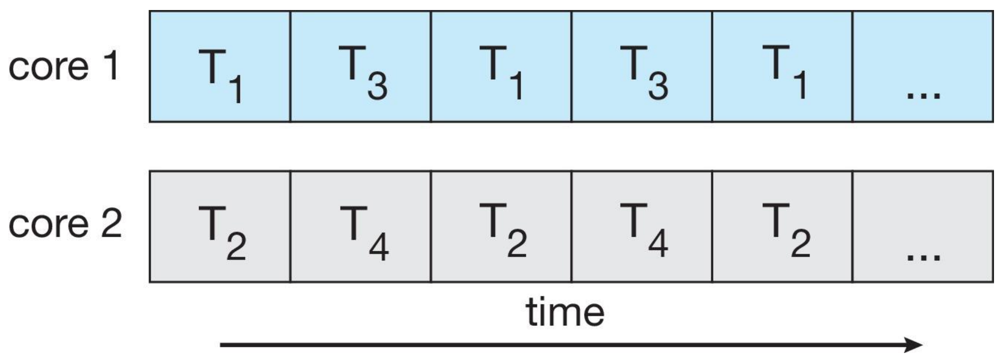
- In response to the need for more computing performance, single-CPU systems evolved into multi-CPU systems.
  - Current trend in system design is to place multiple computing cores on a single processing chip
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** 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:**

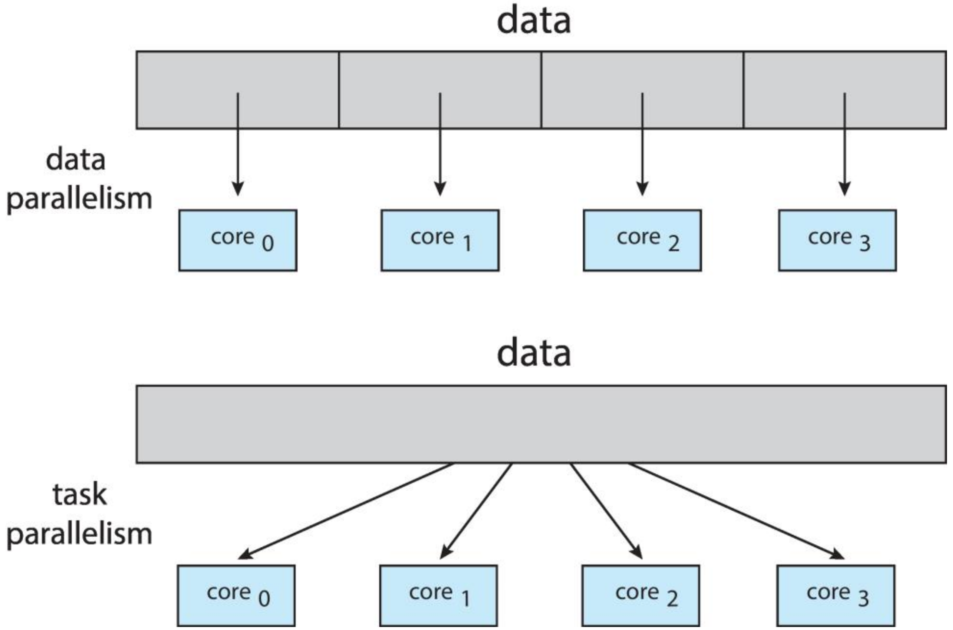


Multicore Programming

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

Multicore Programming

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



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

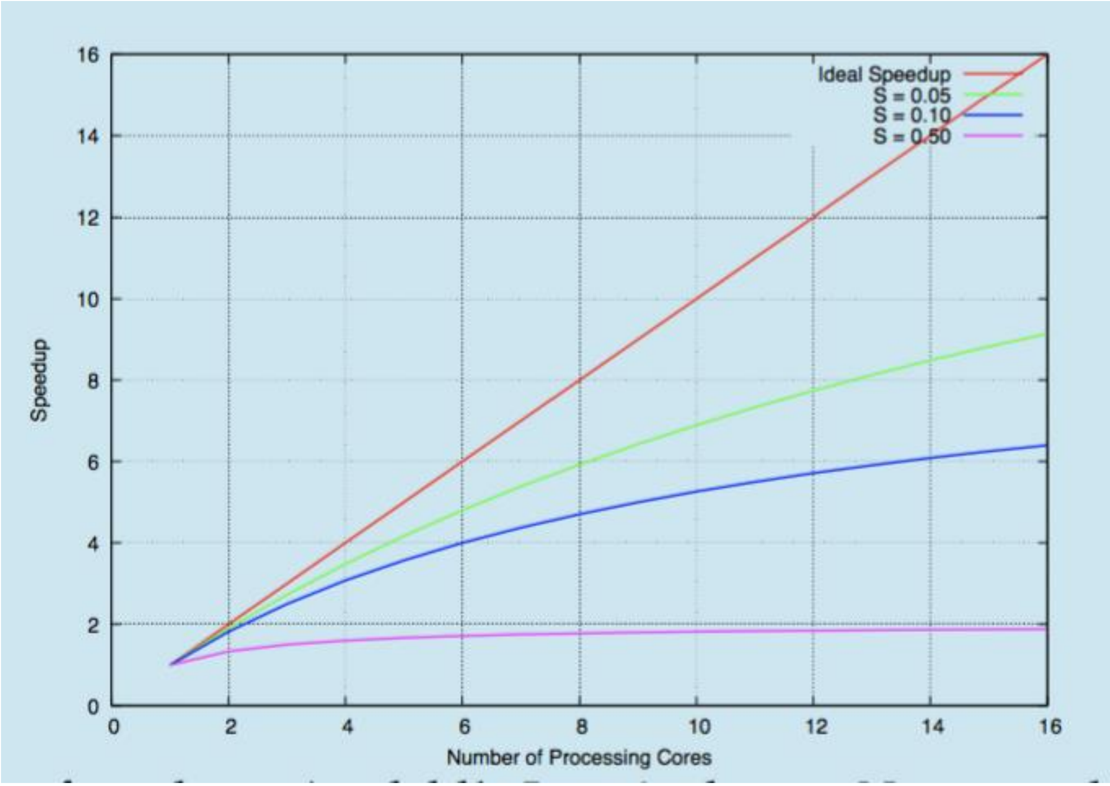
$$speedup \le \frac{1}{S + \frac{(1-S)}{N}}$$

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

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

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

Amdahl's Law



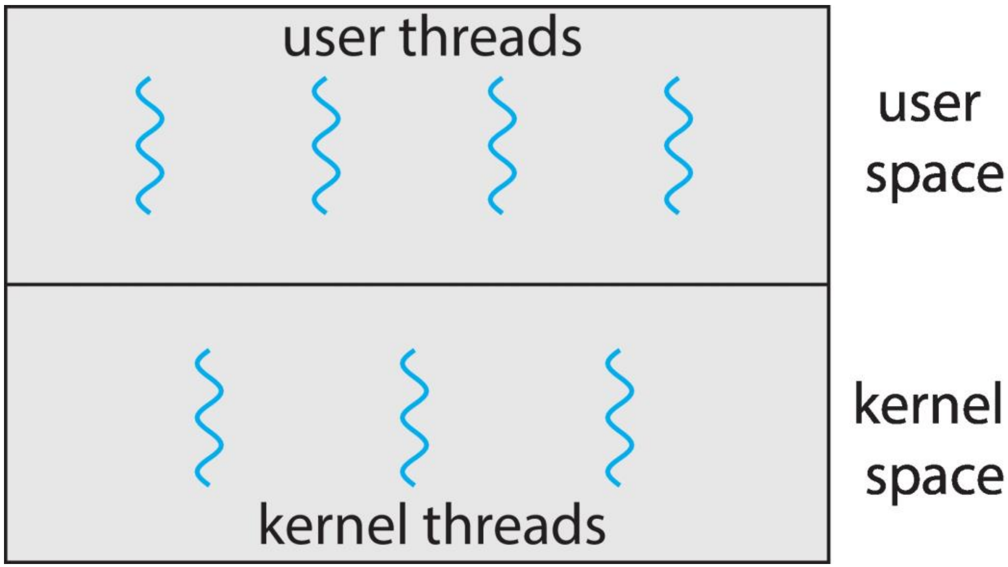
Ideal Speedup
S = 0.05
S = 0.10
S = 0.50
Speedup
Number of Processing Cores
0
2
4
6
8
10
12
14
16

Thread Libraries

- Thread libraries provide programmers with an API for creating and managing threads.
- Thread libraries may be implemented either in user or in kernel space.
  - User space; API functions implemented solely within user space, with no kernel support.
  - Kernal space; involves system calls, and requires a kernel with thread library support.
  - A few well established primary thread libraries
    - POSIX Pthreads - may be provided as either a user or kernel library
    - Win32 threads - provided as a kernel-level library on Windows systems.
    - Java threads – May be Pthreads or Win32 depending on the OS and hardware the JVM is running.

User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- **Kernel threads** - Supported by the Kernel
  - Exists virtually in all general purpose OS:
    - Windows, Linux, Mac OS X, iOS, Android
- Even user threads will ultimately need kernel thread support (Why??)

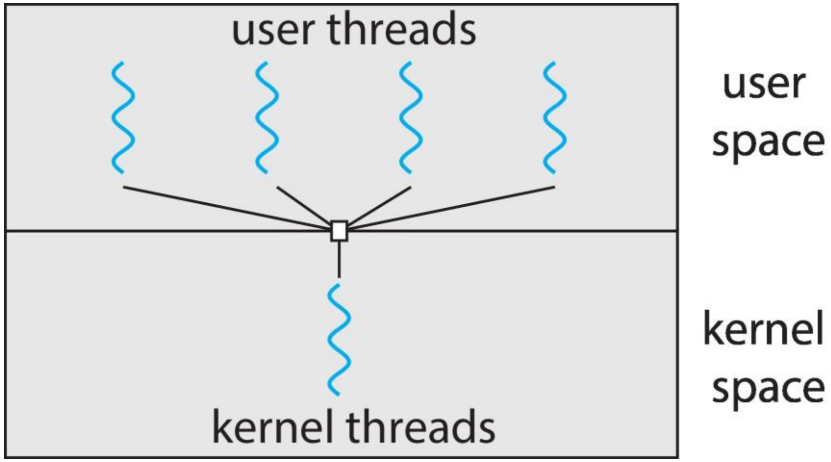


Multithreading Models

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

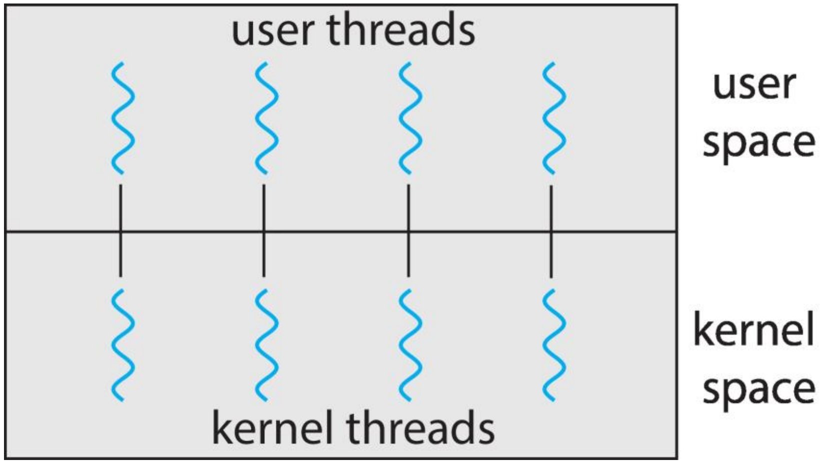
Many-to-One

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



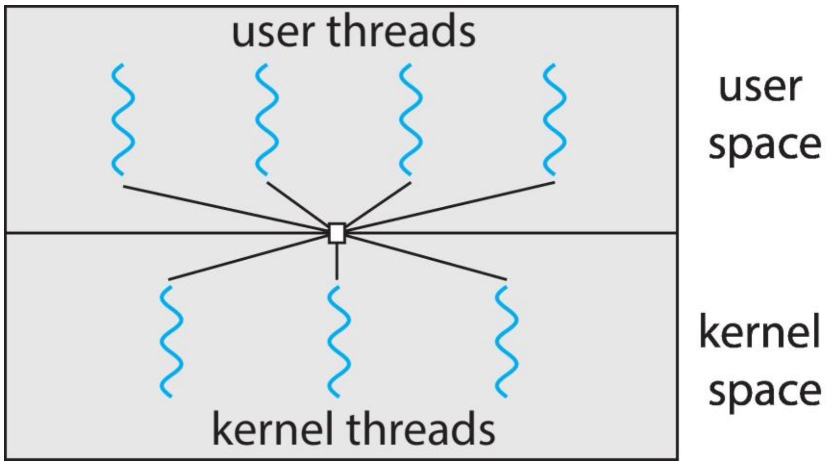
One-to-One

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



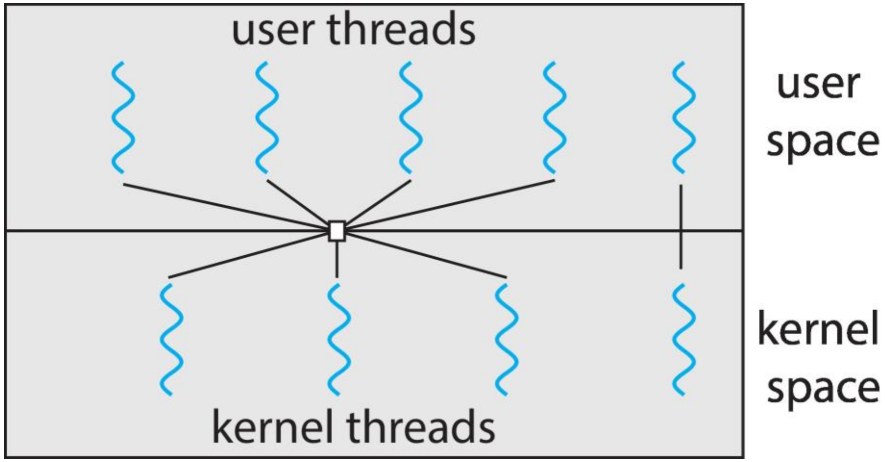
Many-to-Many Model

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



Two-level Model

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



Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard API for thread creation and synchronization
- ***Specification***, not ***implementation***
  - API specifies behavior of the thread library, implementation is up to development of the library
- Example: Sum of N natural numbers
- Global data: Any variable declared globally are shared among all threads of the same process
- Local data: Data local to a function (running in a thread) are stored in thread stack

Pthreads Example: Code

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

Pthreads Example: Code

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

int sum; // global variable shared over threads
```

Pthreads Example: Code

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

int sum; // global variable shared over threads
void *runner (void *param); // threads begin execution in a specified function
```

Pthreads Example: Code

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

int sum; // global variable shared over threads
void *runner (void *param); // threads begin execution in a specified function
int main(int argc, char *argv[]){









}
```

Pthreads Example: Code

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

int sum; // global variable shared over threads
void *runner (void *param); // threads begin execution in a specified function
int main(int argc, char *argv[]){
        pthread_t tid; \\ declares the identifier for the thread







}
```

Pthreads Example: Code

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

int sum; // global variable shared over threads
void *runner (void *param); // threads begin execution in a specified function
int main(int argc, char *argv[]){
        pthread_t tid; \\ declares the identifier for the thread
        pthread attr t attr; \\ set of thread attributes




}
```

Pthreads Example: Code

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

int sum; // global variable shared over threads
void *runner (void *param); // threads begin execution in a specified function
int main(int argc, char *argv[]){
        pthread_t tid; \\ declares the identifier for the thread
        pthread attr t attr; \\ set of thread attributes
        pthread attr init(&attr); \\ set the default attributes of the thread



}
```

Pthreads Example: Code

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

int sum; // global variable shared over threads
void *runner (void *param); // threads begin execution in a specified function
int main(int argc, char *argv[]){
        pthread_t tid; \\ declares the identifier for the thread
        pthread attr t attr; \\ set of thread attributes
        pthread attr init(&attr); \\ set the default attributes of the thread
        pthread create(&tid, &attr, runner, argv[1]); \\ create the thread


}
```

Pthreads Example: Code

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

int sum; // global variable shared over threads
void *runner (void *param); // threads begin execution in a specified function
int main(int argc, char *argv[]){
        pthread_t tid; \\ declares the identifier for the thread
        pthread attr t attr; \\ set of thread attributes
        pthread attr init(&attr); \\ set the default attributes of the thread
        pthread create(&tid, &attr, runner, argv[1]); \\ create the thread
        pthread join(tid,NULL); \\ wait for the thread to exit

}
```

Pthreads Example: Code

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

int sum; // global variable shared over threads
void *runner (void *param); // threads begin execution in a specified function
int main(int argc, char *argv[]){
        pthread_t tid; \\ declares the identifier for the thread
        pthread attr t attr; \\ set of thread attributes
        pthread attr init(&attr); \\ set the default attributes of the thread
        pthread create(&tid, &attr, runner, argv[1]); \\ create the thread
        pthread join(tid,NULL); \\ wait for the thread to exit
        printf("sum = %d\n",sum);
}
```

Pthreads Example: Code

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

int sum; // global variable shared over threads
void *runner (void *param); // threads begin execution in a specified function
/* The thread will execute in this function */
void *runner(void *param) {
        int i, upper = atoi(param);
        sum = 0;
        for (i = 1; i <= upper; i++)
                sum += i;
                pthread exit(0);  \\ thread terminates
}
```

Pthreads Example: Code

- Growing dominance of multicore systems, writing programs containing several threads has become common.
- Simple method for waiting on several threads using the **pthread_join()** function is to enclose the operation within a simple for loop

```
#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);
```

OpenMP

- Collection 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

```c
#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;
}
```

End of Chapter 4