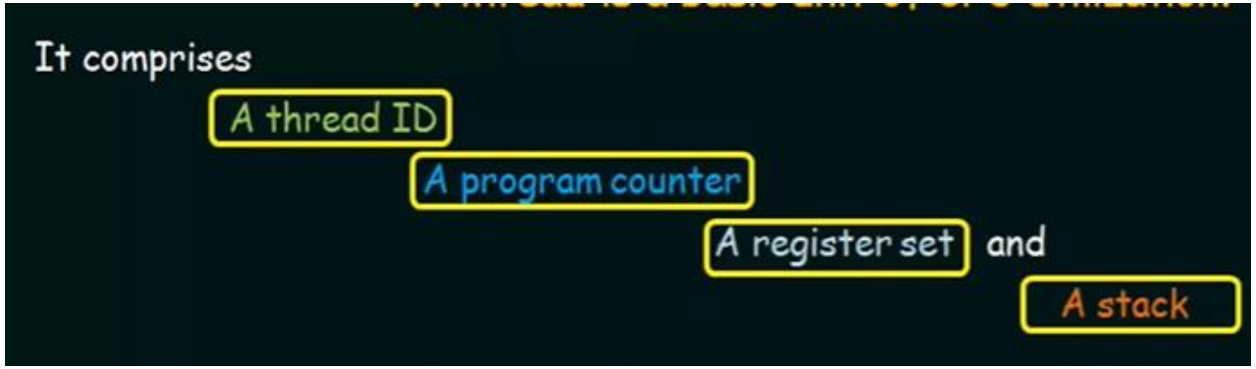
# Chapter 3. Threads and Concurrency

Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. (2018). *Operating System Concepts* (10th ed.). Wiley.

# Contents

# 1. Overview

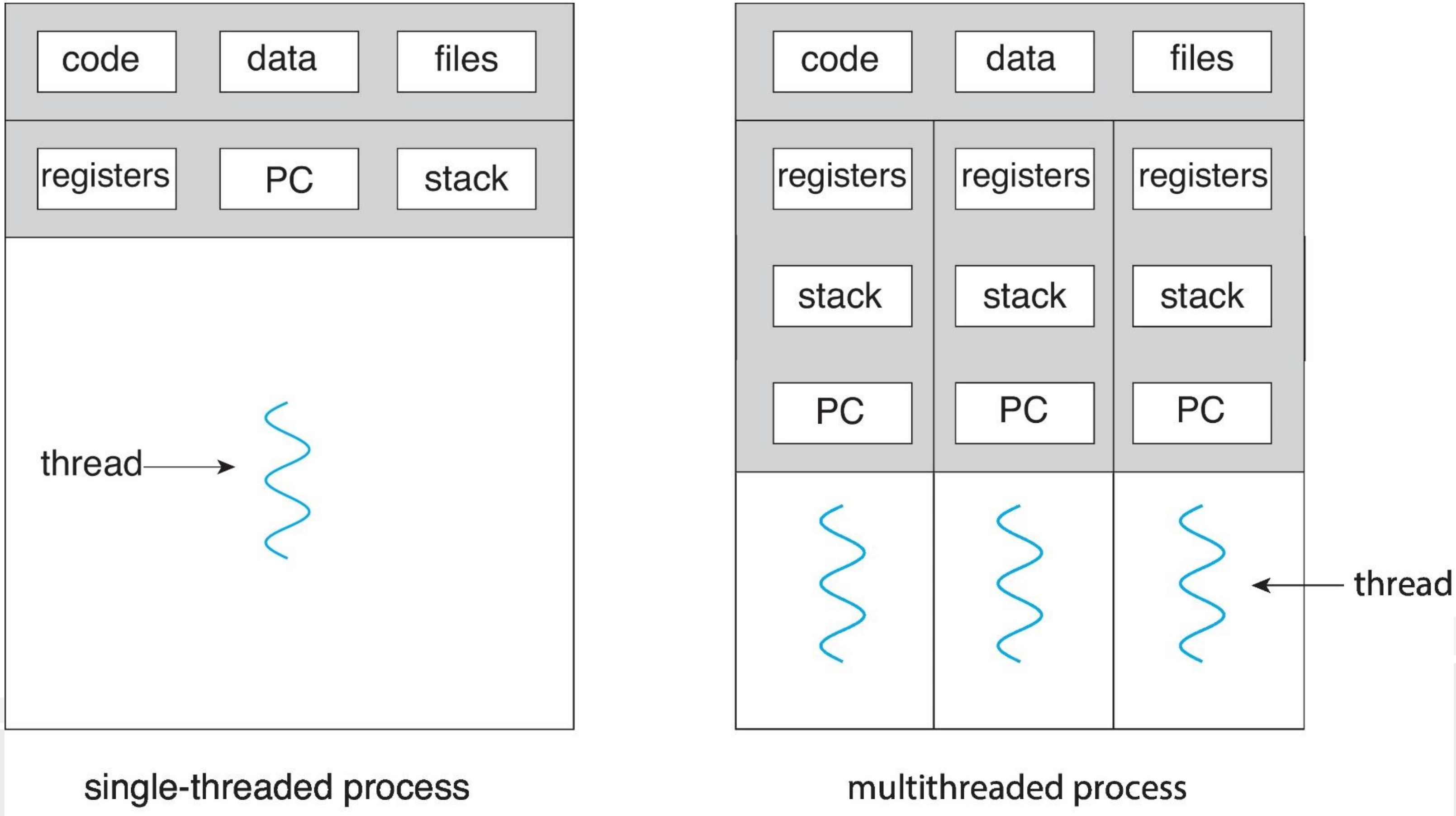
- A thread is a **basic unit of CPU utilization**; it comprises a thread ID, a program counter (PC), a register set, and a stack.




- It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.
- A traditional process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.

# 1. Overview

- Single and multithreaded processes

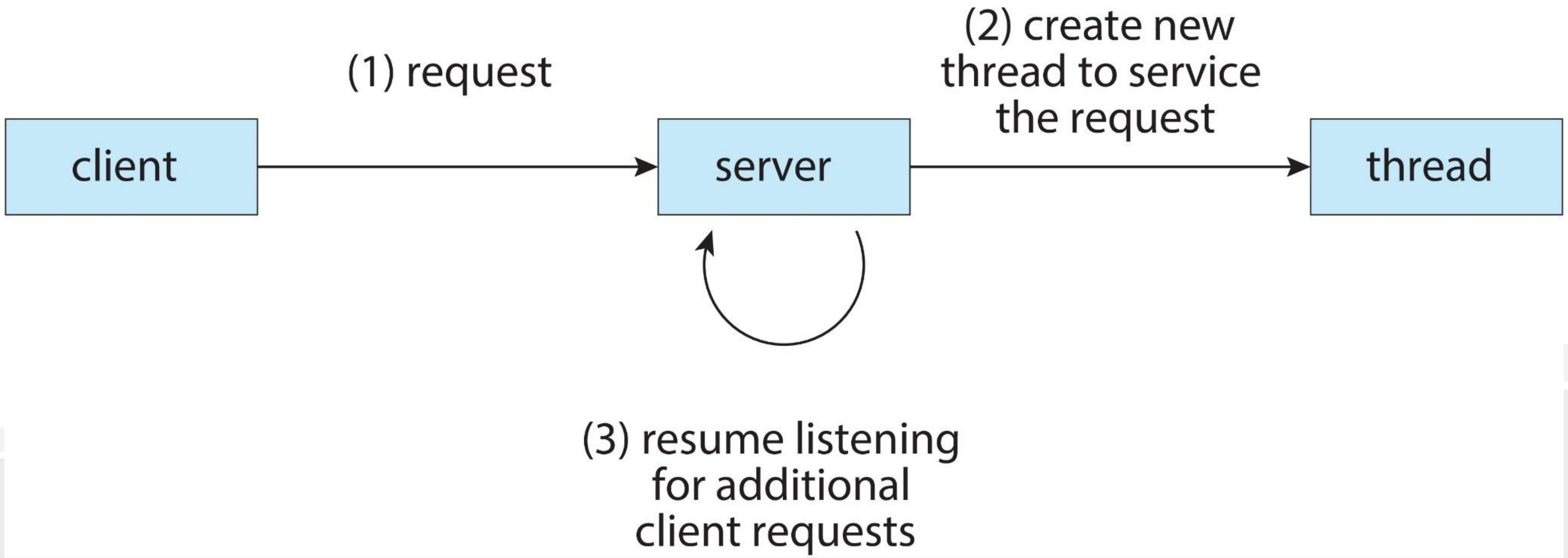


# 1. Overview

- Most modern applications are multithreaded.
- 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

# 1. Overview

- Multithreaded server architecture
  - The server will create a separate thread that listens for client requests.
  - When a request is made, the server creates a new thread to service the request and resumes listening for additional requests.



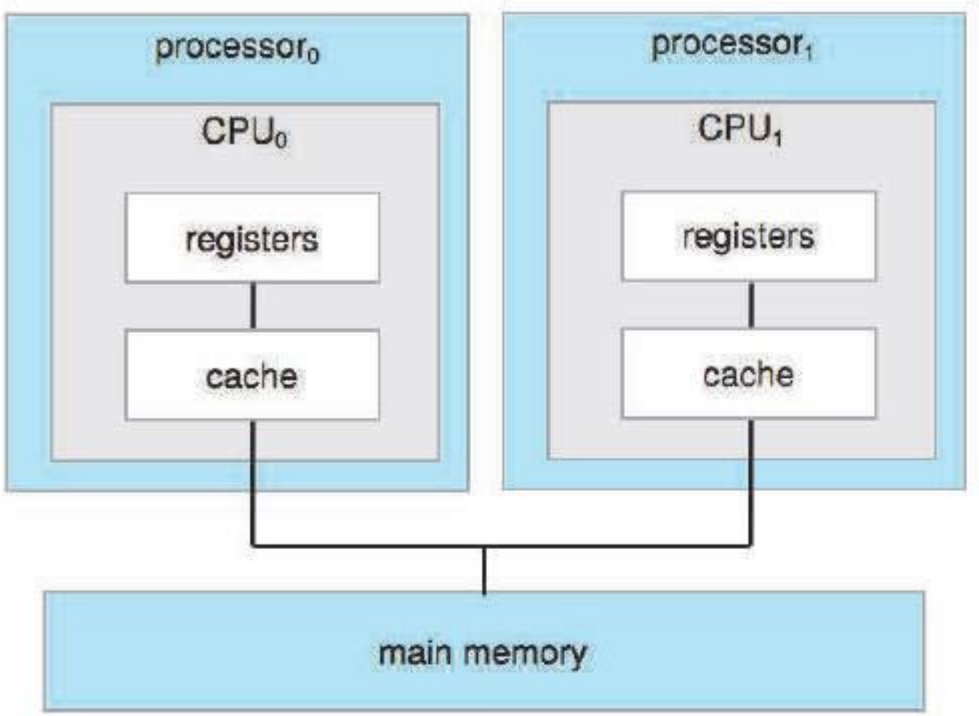
# 1. Overview

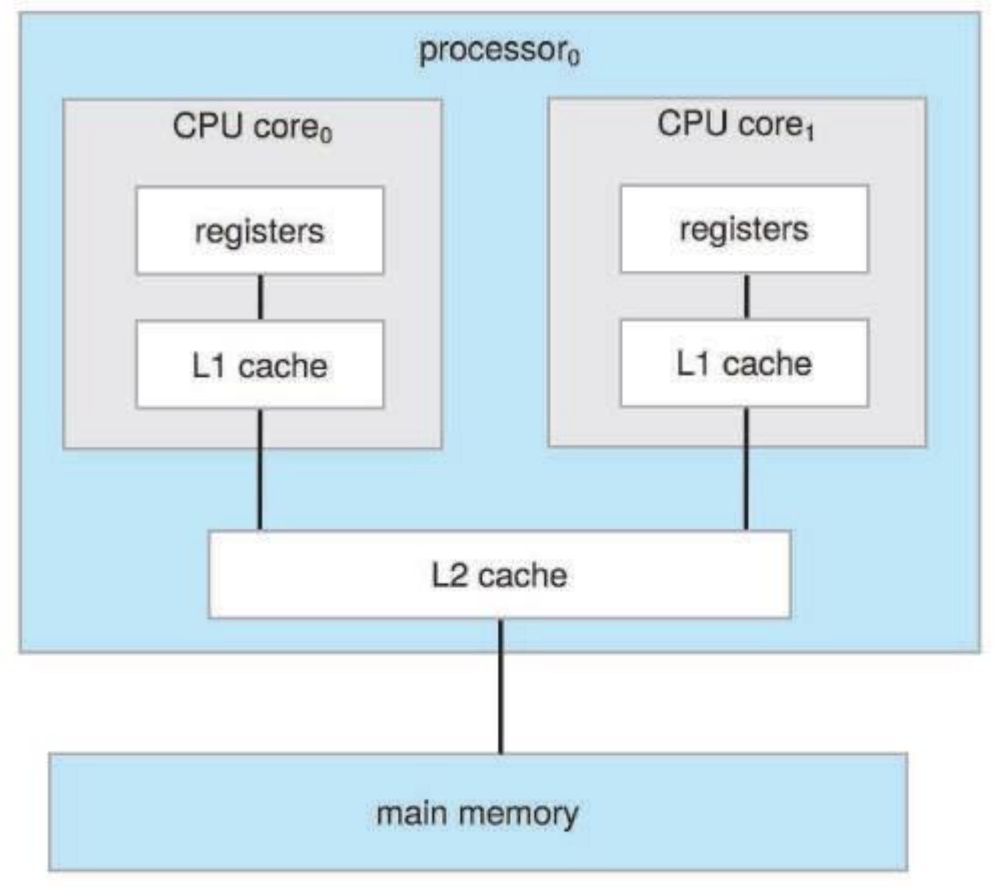
- Benefits
  - **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
  - **Resource Sharing** – threads share resources of process, easier than shared memory or message passing
  - **Economy** – cheaper than process creation, thread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

# 2. Multicore Programming

- Most modern computers are **multiprocessor systems**.
- The definition of multiprocessor has evolved over time and now includes **multicore systems**, in which multiple computing cores reside on a single chip.
- Multicore systems can be more efficient than multiple chips with single cores because on-chip communication is faster than between-chip communication.

# 2. Multicore Programming



# 2. Multicore Programming

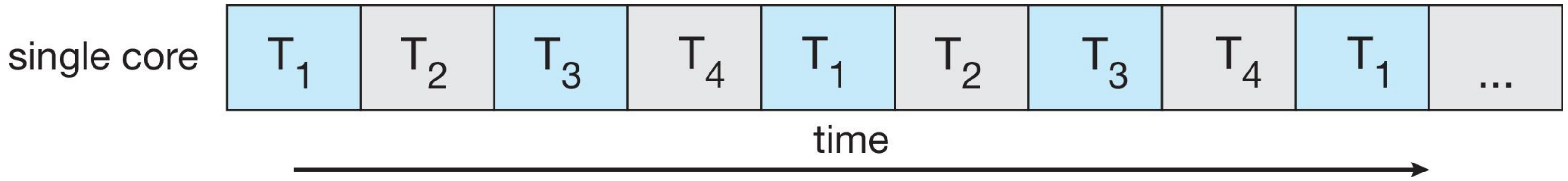
- Multicore or multiprocessor systems puts pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging
- **Parallelism** implies a system can perform more than one task simultaneously.
- **Concurrency** supports more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

# 2. Multicore Programming

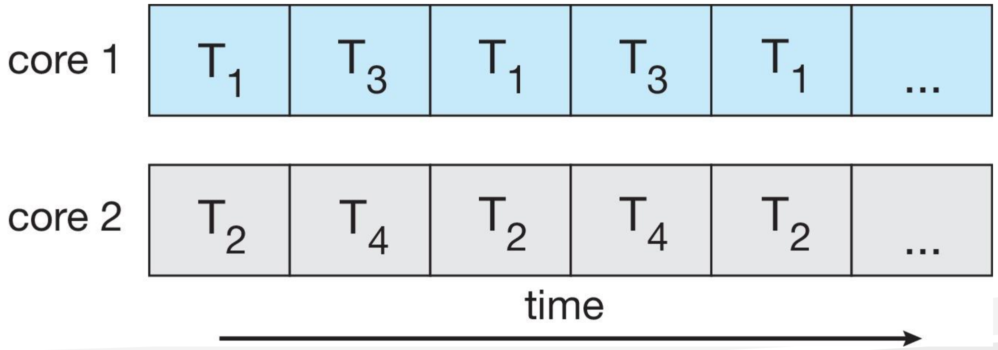
- On a system with a single computing core, **concurrency** means that the execution of the threads will be interleaved over time, because the processing core is capable of executing only one thread at a time.
- On a system with multiple cores, however, concurrency means that some threads can run in parallel, because the system can assign a separate thread to each core.
  - These kinds of execution are called Parallelism
- **Parallelism** implies a system can perform more than one task simultaneously.
- **Concurrency** supports more than one task making progress

# 2. Multicore Programming

- Concurrent execution on single-core system:

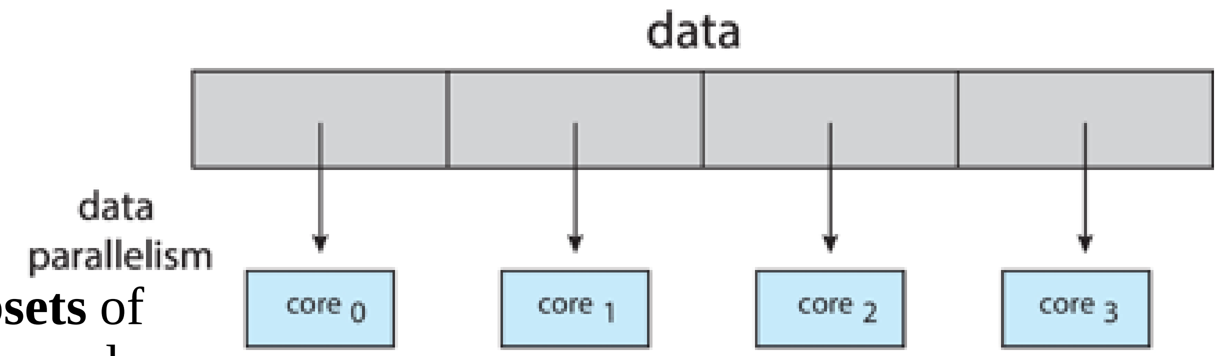


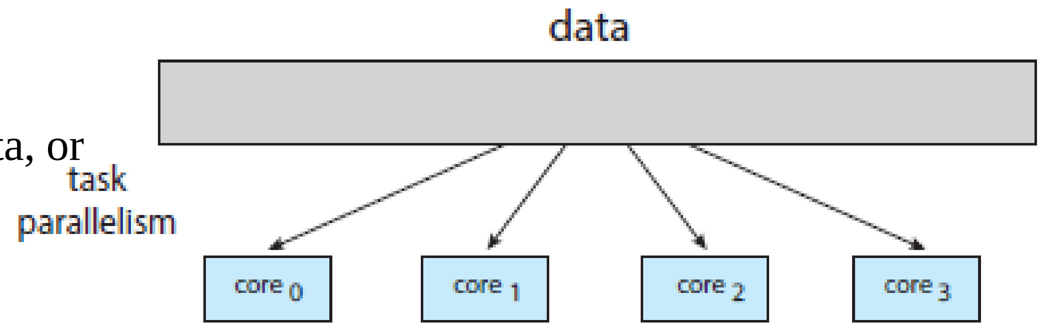

- Parallelism on a multi-core system:



# 2. Multicore Programming

- Types of parallelism
  - Data parallelism
  - Task parallelism
- **Data parallelism** focuses on distributing **subsets** of the same data across multiple computing cores and performing **the same operation** on each core.
- **Task parallelism** involves distributing not data but tasks (threads) across multiple computing cores.
  - Each thread is performing a unique operation.
  - Different threads may be operating on the same data, or they may be operating on different data



# 2. Multicore Programming

- Amdahl's Law
  - Amdahl's Law is a formula that identifies potential performance gains from adding additional computing cores to an application that has both serial (nonparallel) and parallel components. If $S$ is the portion of the application that must be performed serially on a system with $N$ processing cores, the formula appears as follows:

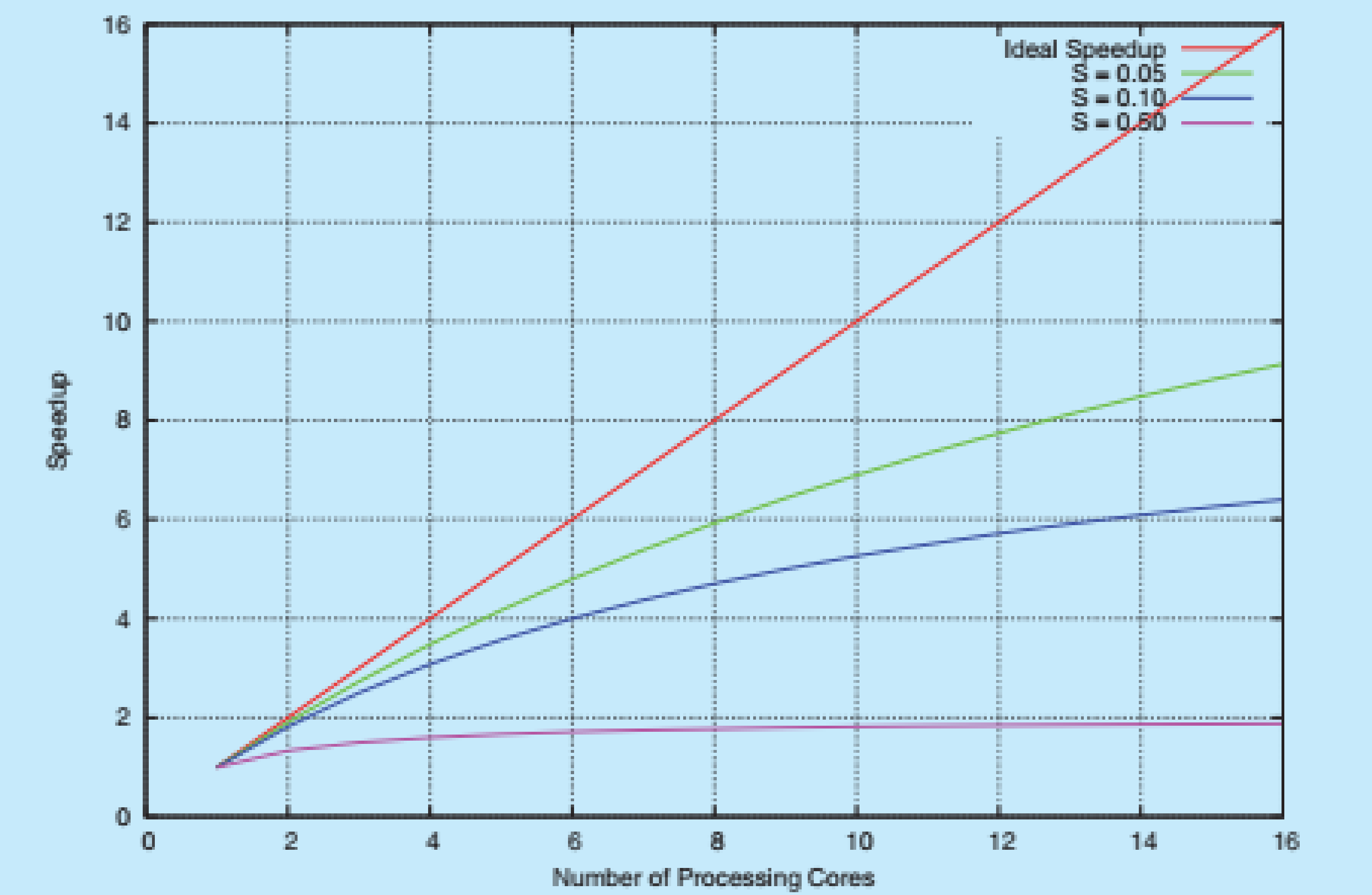
$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

  - As an example, assume we have an application that is 75 percent parallel and 25 percent serial. If we run this application on a system with two processing cores ($N = 2$), we can get a speedup of 1.6 times. If we add two additional cores ($N = 4$), the speedup is 2.28 times.
  - As N approaches infinity, speedup approaches 1 / S.
  - Serial portion of an application has disproportionate effect on performance gained by adding additional cores.

# 2. Multicore Programming

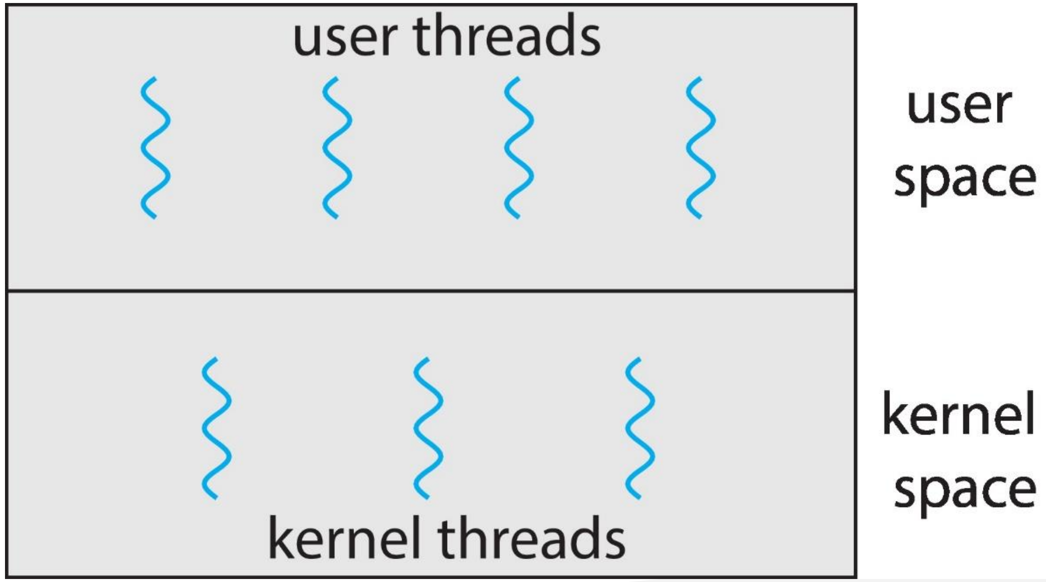
- Amdahl's Law

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

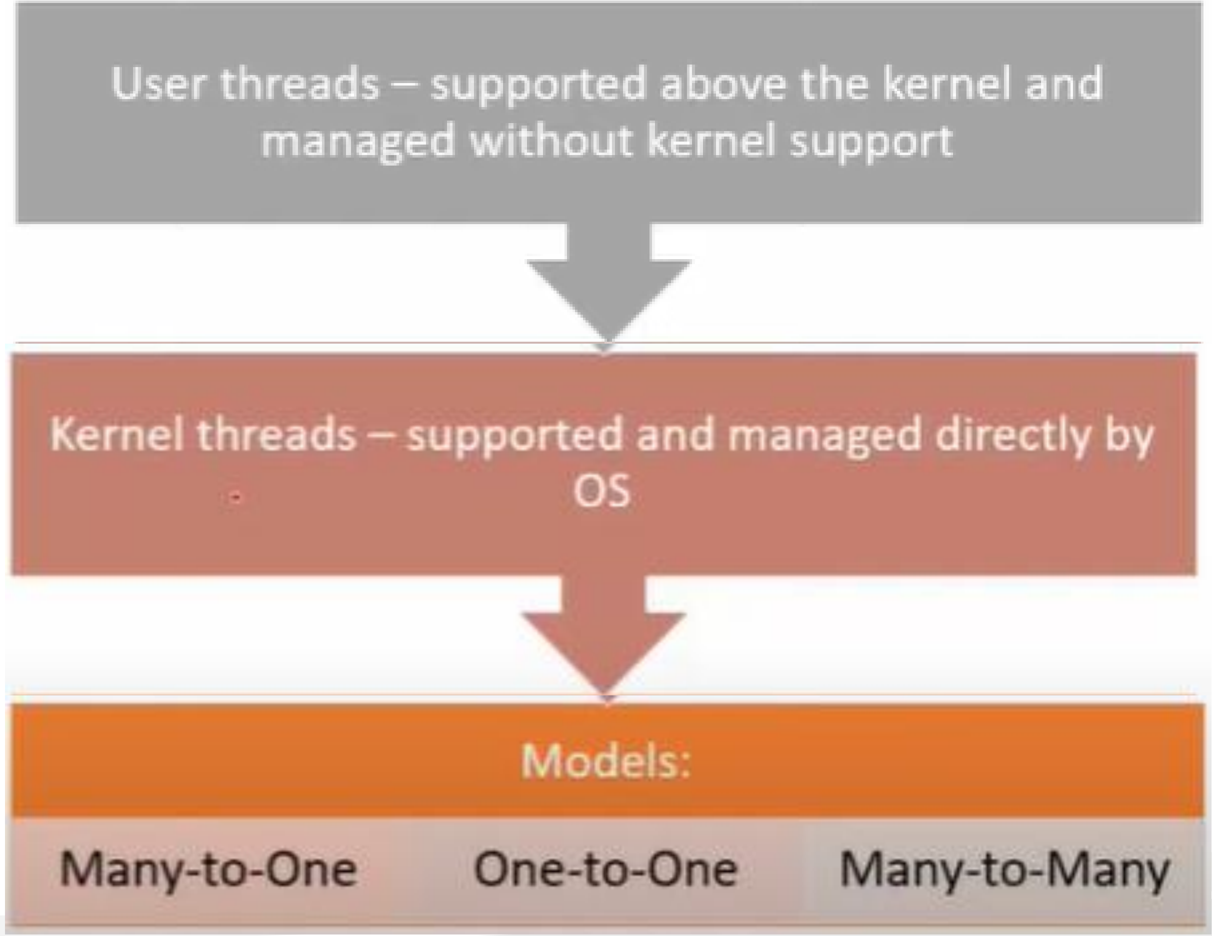


# 3. Multithreading Models

- Support for threads may be provided either at the user level, for **user threads**, or by the kernel, for **kernel threads**.
  - User threads are supported above the kernel and are managed without kernel support.
  - Kernel threads are supported and managed directly by the operating system.
    - Virtually all contemporary operating systems—including Windows, Linux, and macOS— support kernel threads.
  - Ultimately, a relationship must exist between user threads and kernel threads

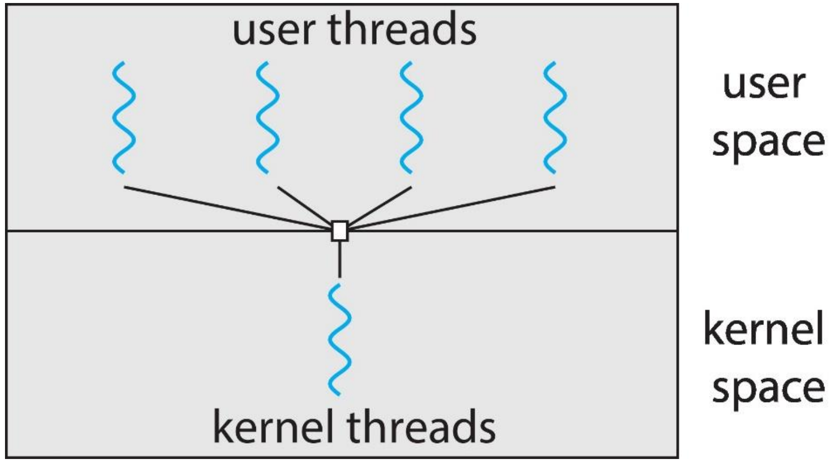


# 3. Multithreading Models



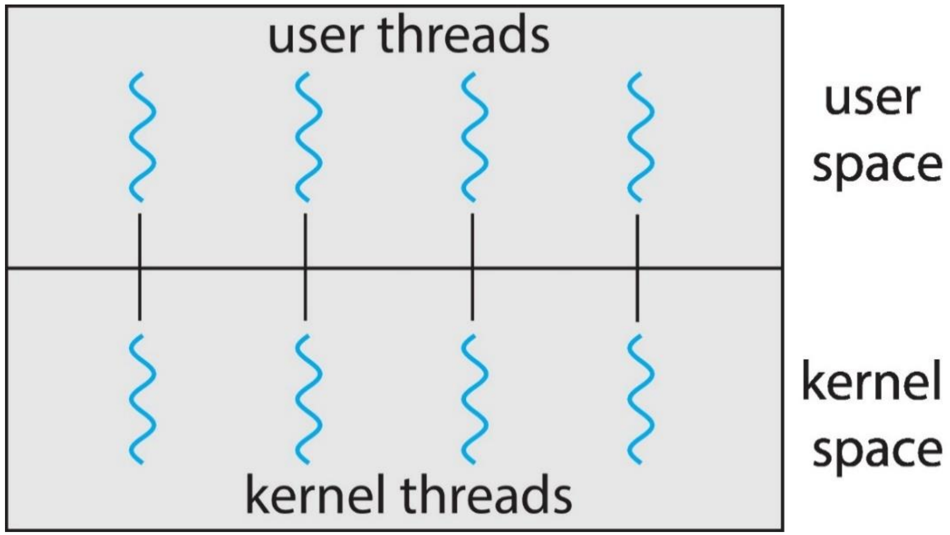
# 3. Multithreading Models

- Many-to-One Model
  - Many user-level threads mapped to single kernel thread.
  - Thread management is done by the thread library in user space, so it is efficient.
  - The entire process will block if a thread makes a blocking system call.
  - Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time.
  - Few systems currently use this model.
    - Solaris Green Threads, Since version 1.3, JVM is no longer implemented with green threads for any platform.
    - GNU Portable Thread



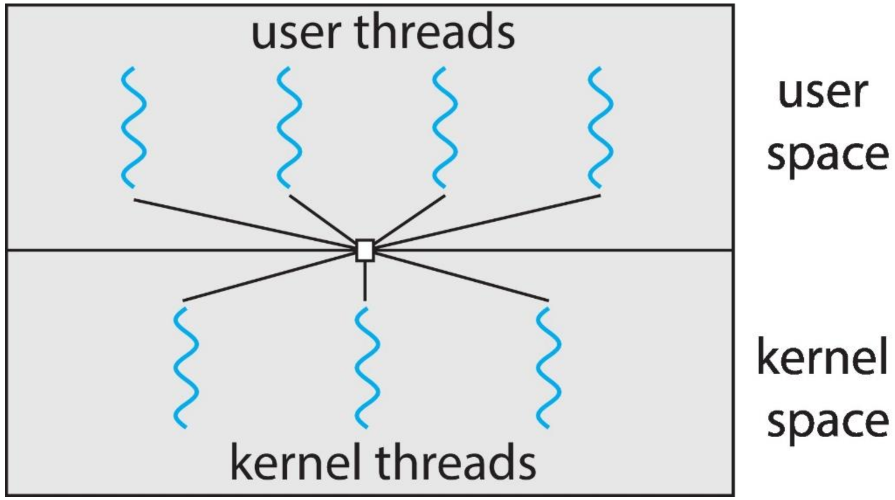
# 3. Multithreading Models

- One-to-One Model
  - 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



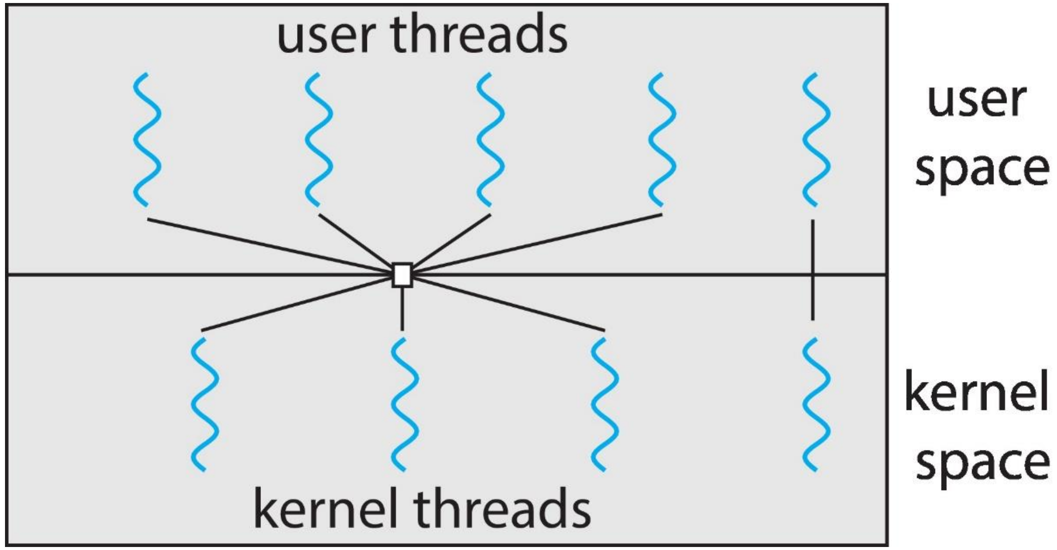
# 3. Multithreading Models

- 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



# 3. Multithreading Models

- Two-level Model
  - Similar to Many-to-Many model, except that it allows a user thread to be bound to a kernel thread.
  - IRIX, HP-UX, and Tru64 UNIX use the two-tier model, as did Solaris prior to Solaris 9



# 4. Thread Libraries

- Thread library provides programmers with API for creating and managing threads.
- Two primary ways of implementing
  - Library entirely in user space
    - Code and data structures for the library exist in user space.
    - This means that invoking a function in the library results in a local function call in user space and not a system call.
  - Kernel-level library supported by the OS
    - Code and data structures for the library exist in kernel space.
    - Invoking a function in the API for the library typically results in a system call to the kernel.

# 4. Thread Libraries

- There are 3 main thread libraries in use today:
  - POSIX Pthreads - may be provided as either a user or kernel library, as an extension to the POSIX standard.
  - Win32 threads - provided as a kernel-level library on Windows systems.
  - Java threads - Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either Pthreads or Win32 threads depending on the system.

# Pthreads

- Pthreads, the threads extension of the POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- May be provided as either a user-level or a kernel-level library
- Specification, not implementation
  - API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Linux & Mac OS X)

# Pthreads Example

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

#include <stdlib.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 */

   /* set the default attributes of the thread */
   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 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);
}
```

- The program has two threads: the initial (or parent) thread in main() and the summation (or child) thread performing the summation operation in the runner() function.
- This program follows the thread create/join strategy, whereby after creating the summation thread, the parent thread will wait for it to terminate by calling the pthread join() function.
- The summation thread will terminate when it calls the function pthread exit(). Once the summation thread has returned, the parent thread will output the value of the shared data sum.

# Pthreads Example

- Pthread code for joining ten 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);
```

# Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
  - On a UNIX system the JVM normally uses PThreads and on a Windows system it normally uses windows threads.
- 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) { }
```