# CS420: Operating Systems

# Threads

James Moscola
Department of Engineering & Computer Science
York College of Pennsylvania

YORK COLLEGE OF PENNSYLVANIA

# Threads

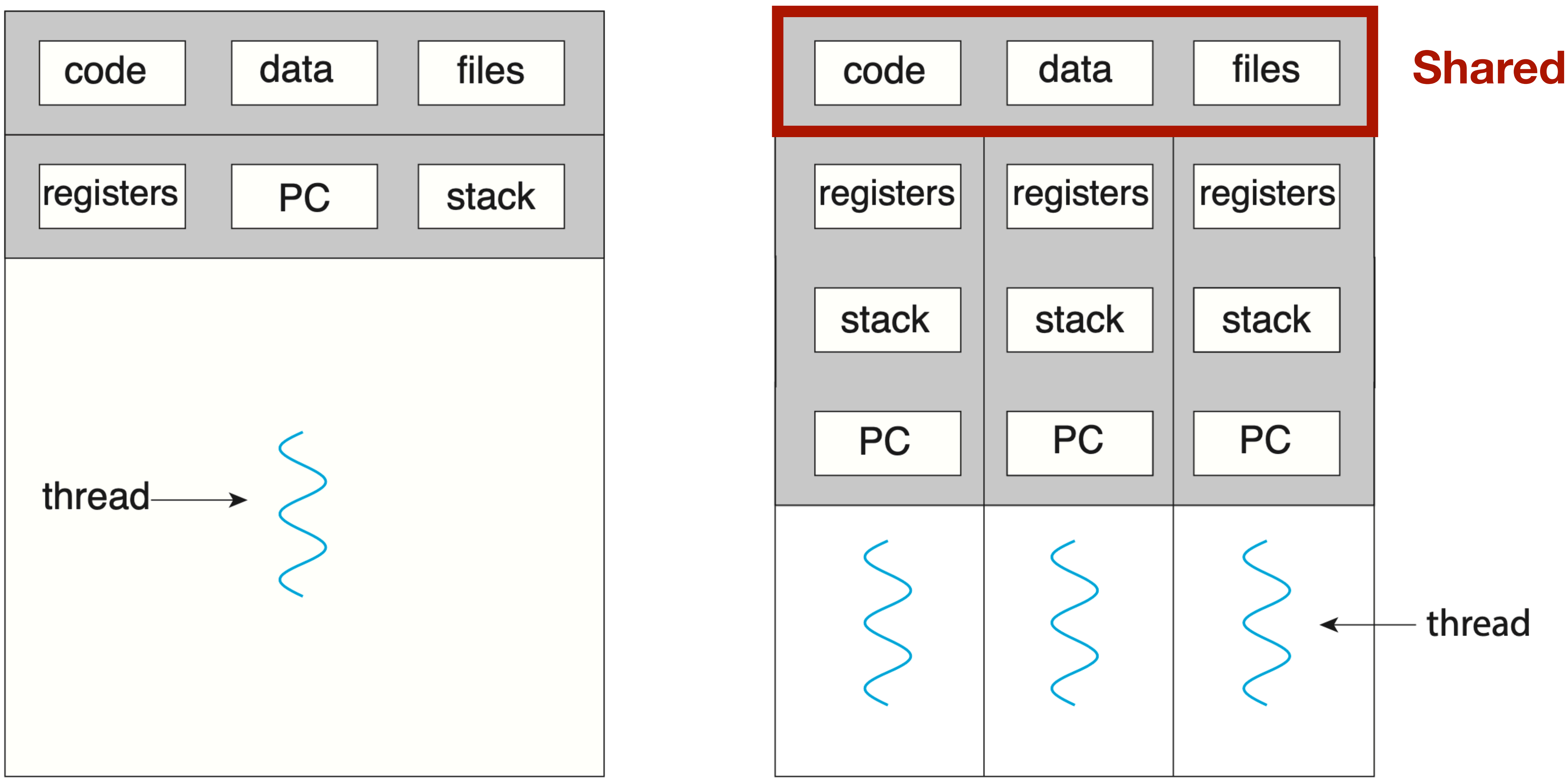
- **A thread is a basic unit of processing**
  - Has the following components:
    - Thread ID
    - Program counter
    - Register set
    - Stack
  - Shares some resources with other threads in same process
    - Code section
    - Data section
    - OS Resources (e.g. open files, signals, etc.)

# Threads

- **A heavyweight process is a process that has a single thread of control**
  - Can only perform a single task at a time

- **A multi-threaded process is a process that has multiple threads of control**
  - Can perform more than one task at a time
    - Render images (e.g. thumbnails for an image library)
    - Fetch data
    - Update display
    - Check spelling

# Single and Multithreaded Processes




single-threaded process

multithreaded process

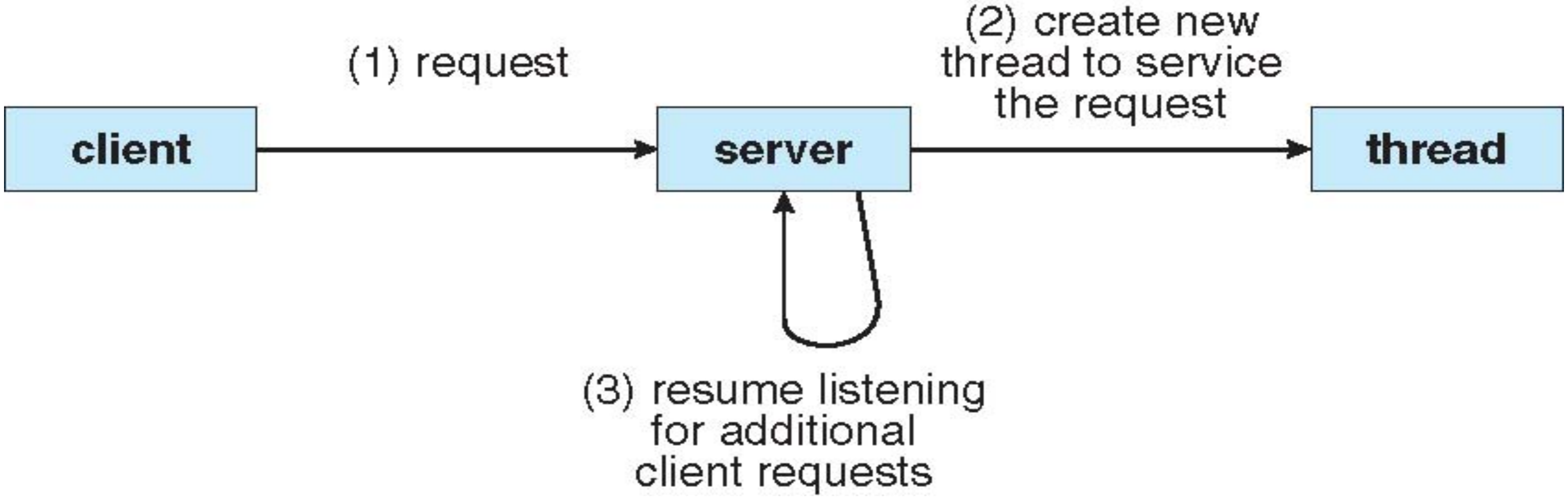
# Thread vs Process

- **Processes**
  - Independent units of execution
  - Each contains its own state information
  - Each contains its own address space
  - Interact with each other through various IPC mechanisms
- **Threads (within the same process)**
  - Share the same state
  - Share the same memory space
  - Share the same variables
  - Can communicate directly through shared variables
  - Share signal handling

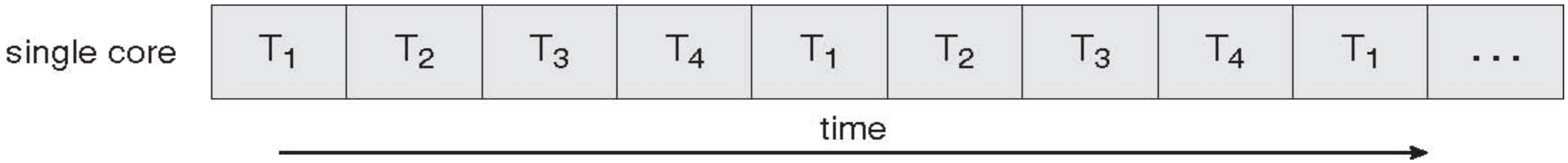
# Benefits of Multithreaded Programming

- **Responsiveness**
  - Interactive applications are more responsive when multithreaded (e.g. a thread for the GUI, another for socket, a third for rendering, etc.)
- **Resource Sharing**
  - Unlike processes, threads share memory (code and data sections) and resources
- **Economy**
  - Since threads share resources, creating threads and switching between them is more efficient than processes
- **Scalability**
  - Multithreading allows for increased parallelism on multicore systems as each thread can run on a different CPU core (kernel threads only)

# Multithreaded Server Architecture

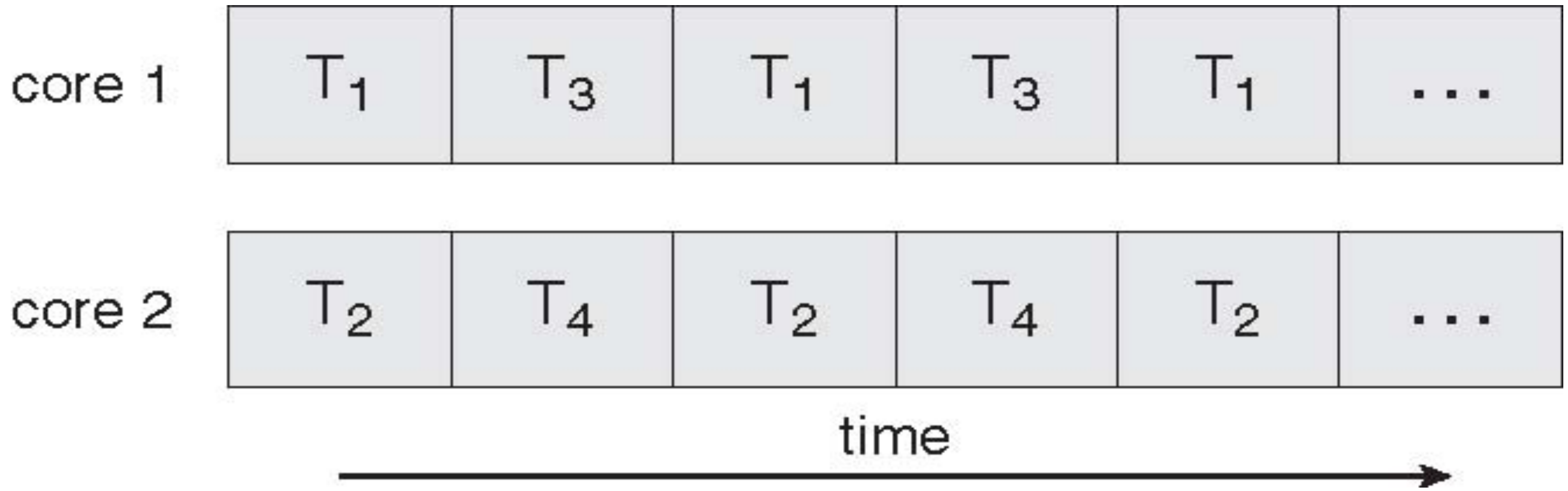


# Concurrent Execution on a Single-core System




- **Only a single thread can execute at a time**
- **Threads are interleaved so each gets time on the processor**

# Parallel Execution on a Multicore System




- **With multiple cores, threads can be divided over the cores and run in parallel**
- **May still interleave threads if not enough cores are available for all of the threads**

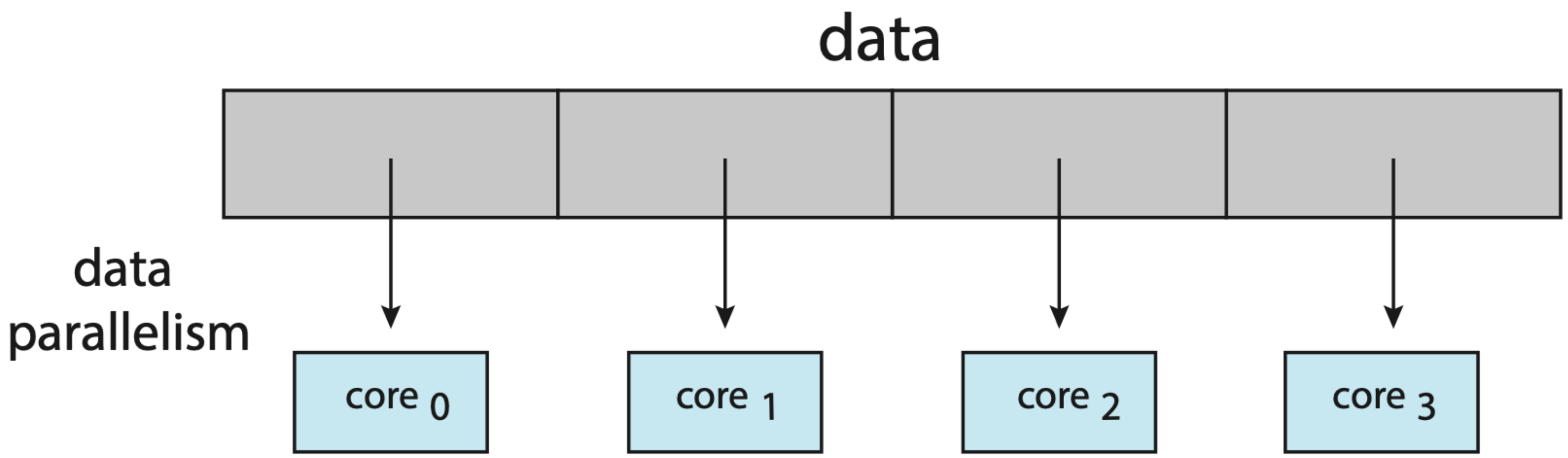
# Multicore Programming

- **The performance of a process can be increased by properly threading the process to take advantage of modern multicore CPUs**

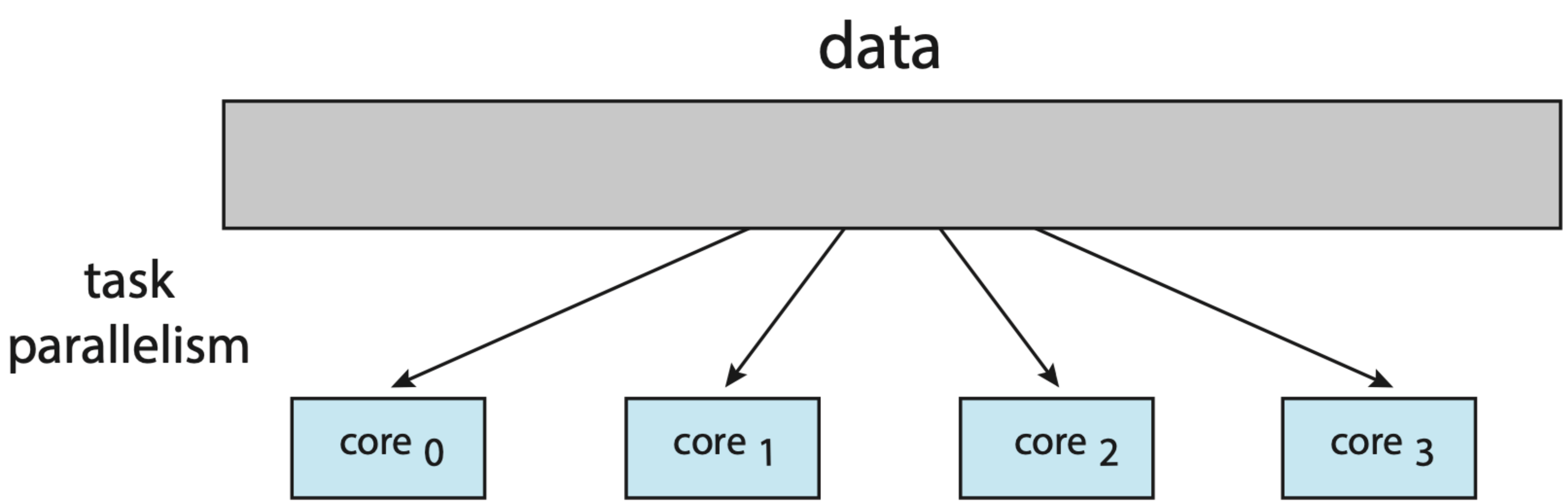
- **Multicore systems have challenges not faced in a single-core/single-threaded environment**
  - **Dividing activities** - How can an application be divided into separate, concurrent tasks?
  - **Balance** - How can those tasks be divided in such a way that each does an equal amount of work?
  - **Data splitting** - Can the data for those task be divided for processing on separate CPU cores?
  - **Data dependency** - Are there data dependencies between different tasks?
  - **Testing and debugging** - What is the best way to debug a multithreaded program with many different execution paths?

# Data Parallelism and Task Parallelism

- **Data Parallelism** - **distribute subsets of the same data across multiple cores**




- **Task Parallelism** - **distribute threads across multiple cores (each performing a unique operation)**



# Thread Support

- **Threads may be supported at different levels of the OS**
  - **User threads**
    - Supported above the kernel
    - Managed and scheduled without kernel support
    - Main user thread libraries currently in use
      POSIX PThreads / Java threads
  - **Kernel threads**
    - Supported by the kernel/operating system
    - Managed and scheduled by the kernel/operating system
    - Most modern operating systems support kernel threads
      (e.g. Windows 2000/XP/…/10/11, Solaris, Linux, macOS)

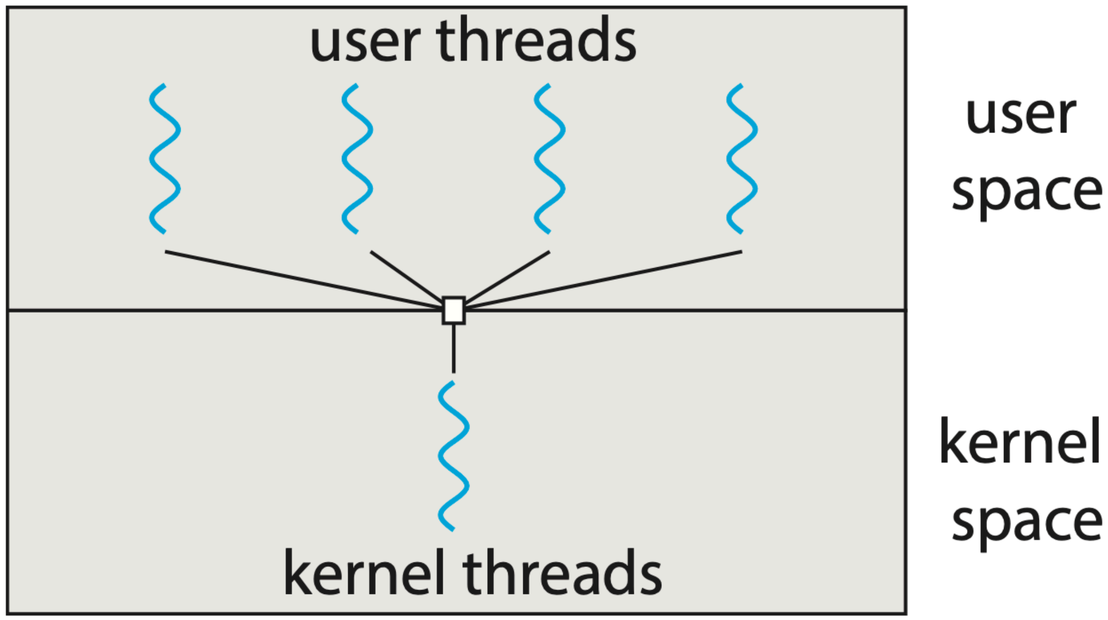
# Multithreading Models

- **There must be a relationship between user level threads and kernel threads**

- **Different models of threading exist to define this relationship**
  - Many-to-One
  - One-to-One
  - Many-to-Many

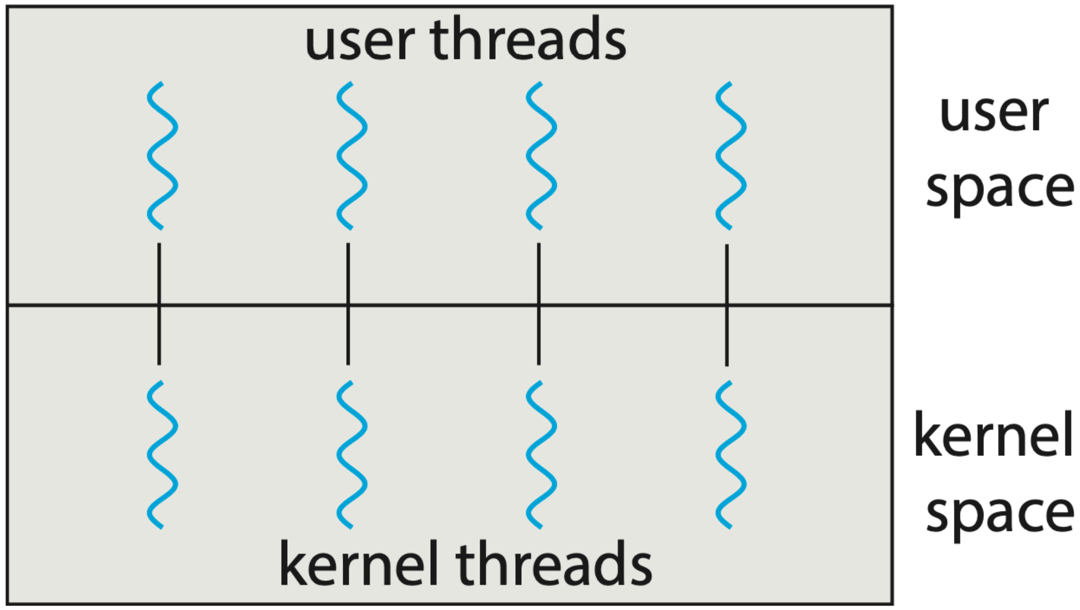
# Many-to-One

- **Many user-level threads mapped to single kernel thread**
  - Examples:  Solaris Green Threads,  GNU Portable Threads
  - Not many systems use this model
- **Thread management is done in user space**
- **Entire process will block if any single thread blocks (no other threads will run)**
- **Unable to run multiple user-level threads in parallel on a multiprocessor system**
  - Not very common anymore



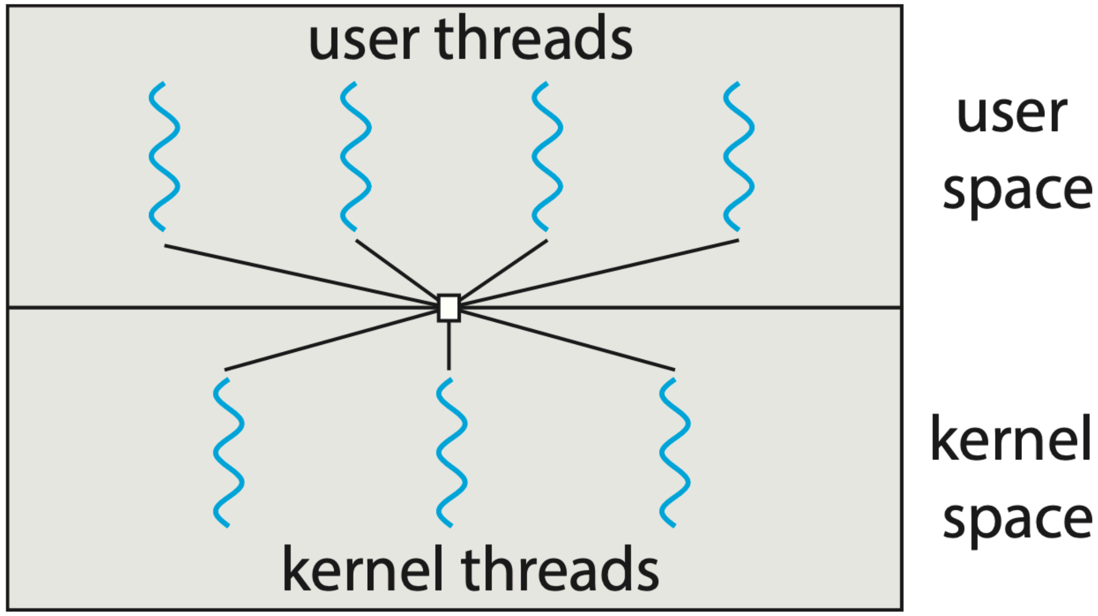
# One-to-One

- **Each user-level thread maps to kernel thread**
  - Examples:  Windows NT/XP/2000, Linux, Solaris 9 and later
- **Allows more concurrency**
  - A thread can run when another thread has made a blocking system call
  - Multiple user-level threads can run in parallel on multiprocessor systems
- **Downside — for each thread created, a corresponding kernel thread must also be created**
  - Large number of kernel threads may degrade system performance



# Many-to-Many Model

- **Many user-level threads are mapped to a smaller number of kernel threads**
  - Avoids blocking of threads when a single thread makes a blocking system call



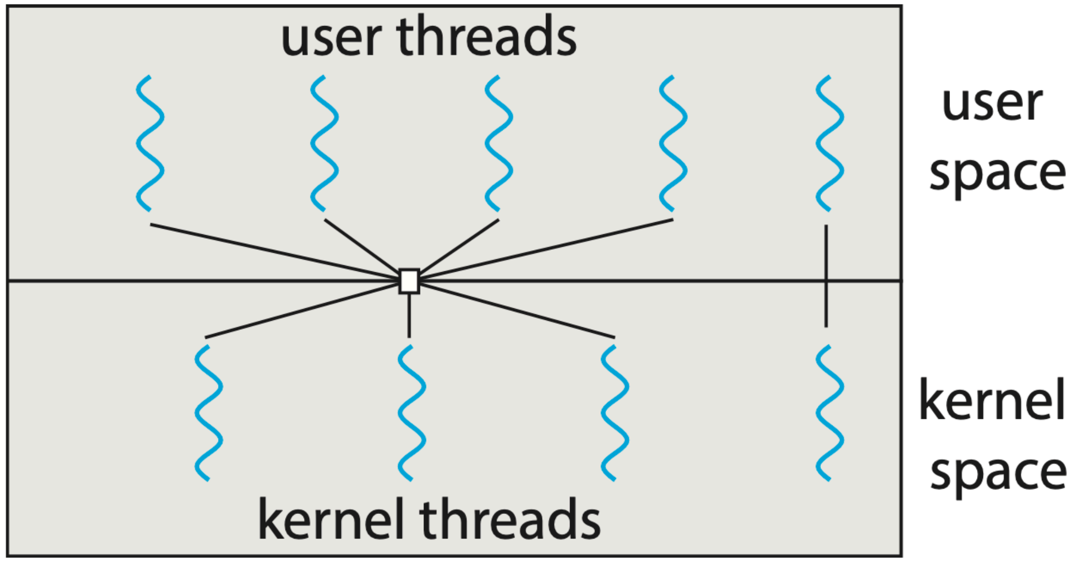

- **Allows the operating system to create a sufficient number of kernel threads**
  - OS may allocate more kernel threads on a machine with more CPU cores

- **Reduces the overhead associated with too many kernel threads as was present in the one-to-one model**

# Two-level Model

- **Similar to the many-to-many model except that it allows a user thread to be bound to a specific kernel thread**

- **Examples include**
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



# Thread Libraries

- **A thread library provides programmer with API for creating and managing threads**

- **Two primary ways of implementing**
  - Library entirely in user space (no kernel support)
  - Kernel-level library supported by the OS

- **Three main thread libraries currently in use**

  (1) POSIX Pthreads - user-level or kernel-level threads for POSIX-compliant systems

  (2) Windows thread library - kernel-level threads for Windows systems

  (3) Java threads - threads created and managed in Java programs (typically mapped to thread library of host system)

# POSIX Pthreads

- **May be provided either as user-level or kernel-level**
- **A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization**
- **API specifies behavior of the thread library, implementation is up to development of the library**
- **Common in UNIX operating systems (Solaris, Linux, macOS)**

# Pthreads Example

```c
#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);
}
```

# Pthreads Example (Cont.)

```
/* 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);
}
```

# 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 API - Multithreaded C Program

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

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 API - Multithreaded C Program (Cont.)

```
/* 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 JVM (Java Virtual Machine)**

- **Typically implemented using the threads model provided by underlying OS**

- **Java threads may be created in two different ways:**
  - Extend the Thread class and override `run()` method
  - Implement the `Runnable` interface

# Java Multithreaded Program

```
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) { }
   }
}
```

# Implicit Threading

- **Creation and management of threads done by compilers and run-time libraries rather than programmers**
  - Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- **Three methods explored**
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch
- **Other methods include Intel Threading Building Blocks (TBB), java.util.concurrent package**

# Thread Pools

- **In applications where threads are repeatedly being created/destroyed thread pools might provide a performance benefit**
  - Example: A server that spawns a new thread each time a client connects to the system and discards that thread when the client disconnects

- **A thread pool is a group of threads that have been pre-created and are available to do work as needed**
  - Threads may be created when the process starts
  - A thread may be kept in a queue until it is needed
  - After a thread finishes, it is placed back into a queue until it is needed again
  - Avoids the extra time needed to spawn new threads when they're needed

# Thread Pools

- **Advantages of thread pools:**
  - Typically faster to service a request with an existing thread than create a new thread (performance benefit)
  - Bounds the number of threads in a process
    - The only threads available are those in the thread pool
    - If the thread pool is empty, then the process must wait for a thread to re-enter the pool before it can assign work to a thread
    - Without a bound on the number of threads in a process, it is possible for a process to create so many threads that all of the system resources are exhausted

# Implicit Threading with OpenMP

- **Set of compiler directives and an API that provides support for parallel programming in shared-memory environments**

- **Identifies parallel regions as blocks of code that may run in parallel**
    - Developers insert compiler directives into their code to define parallel region

# Implicit Threading with OpenMP (Cont.)

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

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

    /* the next bit is automatically parallelized */
    #pragma omp parallel
    {
        printf("I am a parallel region.");
    }


    #pragma omp parallel for
    for (int i = 0; i < 1000; i++) {
        c[i] = a[i] + b[i];
    }

    /* more standard sequential code can go here */

    return 0;
}
```

# Grand Central Dispatch

- **Developed for macOS / iOS (and other Apple operating systems)**
- **Tasks are placed into dispatch queues as program runs**
- **Utilizes multiple dispatch queues**
  - Serial dispatch queue
    - Tasks are removed in FIFO order
    - Once a task is removed, it must be completed prior to next dequeue
  - Concurrent dispatch queue
    - Tasks are removed in FIFO order
    - Multiple tasks may be removed at a time and run in parallel