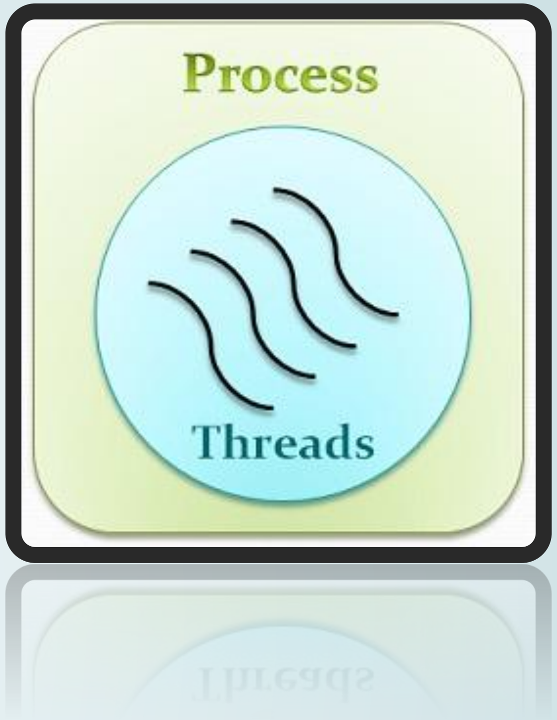


CHAPTER 5: THREADS AND CONCURRENCY

OPERATING SYSTEMS (CS-2006)
FALL 2021, FAST NUCES

anaum.hamid@nu.edu.pk

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

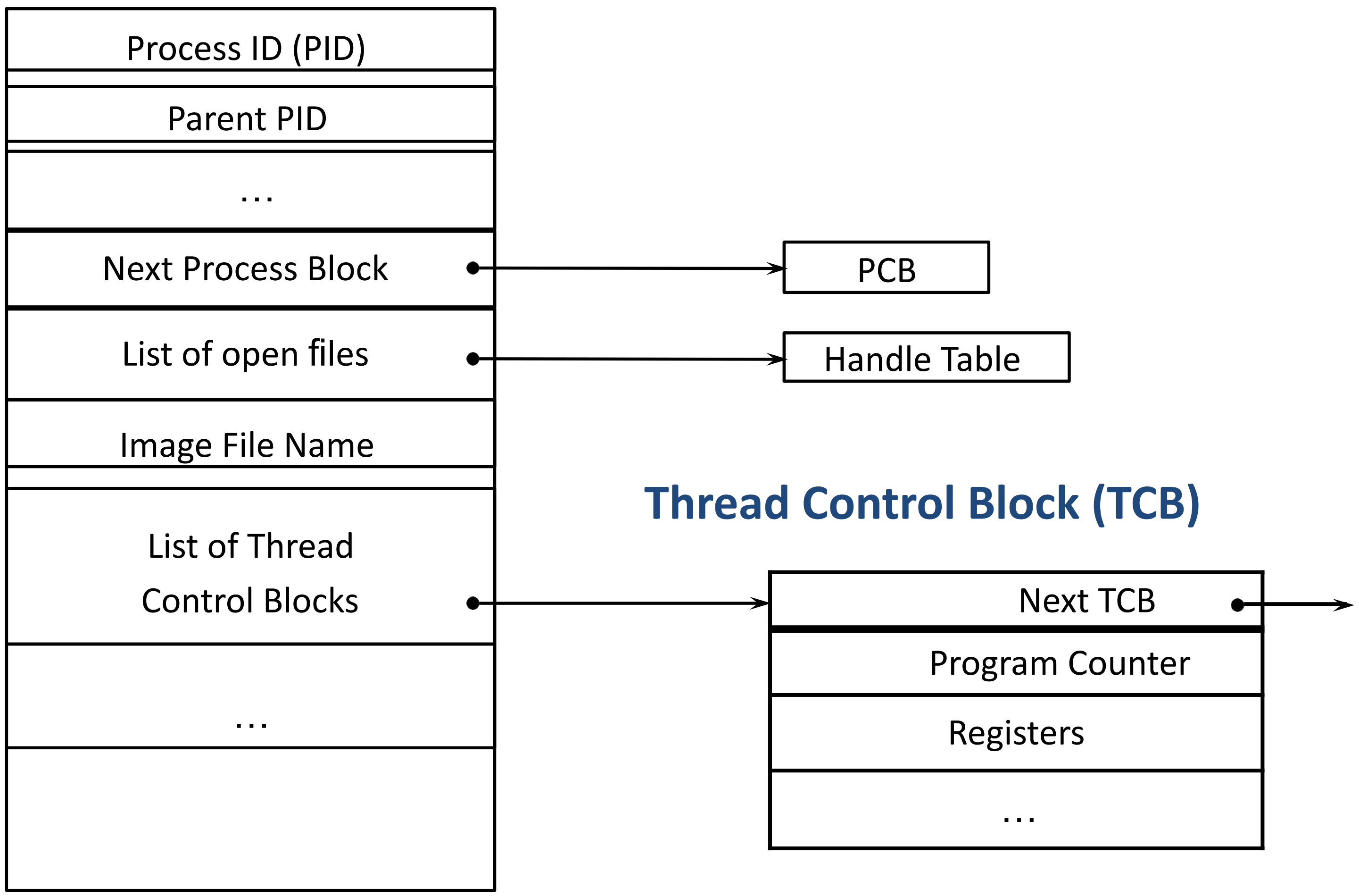
Process Vs. Threads

S.N.	Process	Thread
1.	Process is heavy weight or resource intensive.	Thread is light weight taking lesser resources than a process.
2.	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3.	In multiple processing environments each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4.	If one process is blocked then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, second thread in the same task can run.
5.	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6.	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Control Blocks

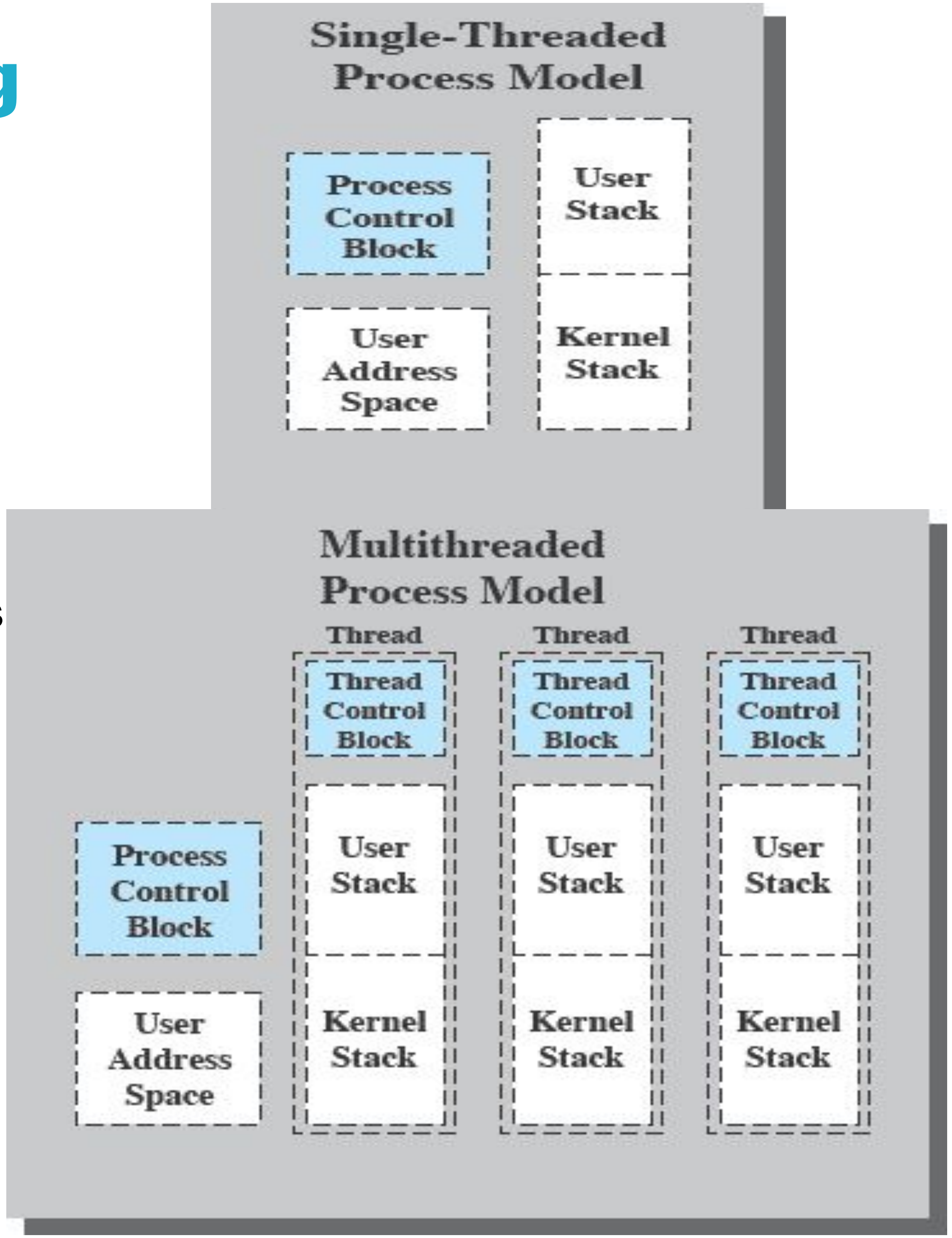
- Information associated with each process: **Process Control Block**
  1. Memory management information
  2. Accounting information
- Information associated with each thread: **Thread Control Block**
  1. Program counter
  2. CPU registers
  3. CPU scheduling information
  4. Pending I/O information

Control Blocks



Single & Multithreading Process

- Each **thread** has
  - An execution state (Running, Ready, etc.)
  - Saved thread context when not running
  - An execution stack
  - Some per-thread static storage for local variables
  - Access to the memory and resources of its process (all threads of a process share this)
- Suspending a process involves suspending all threads of the process
- Termination of a process terminates all threads within the process



Threads

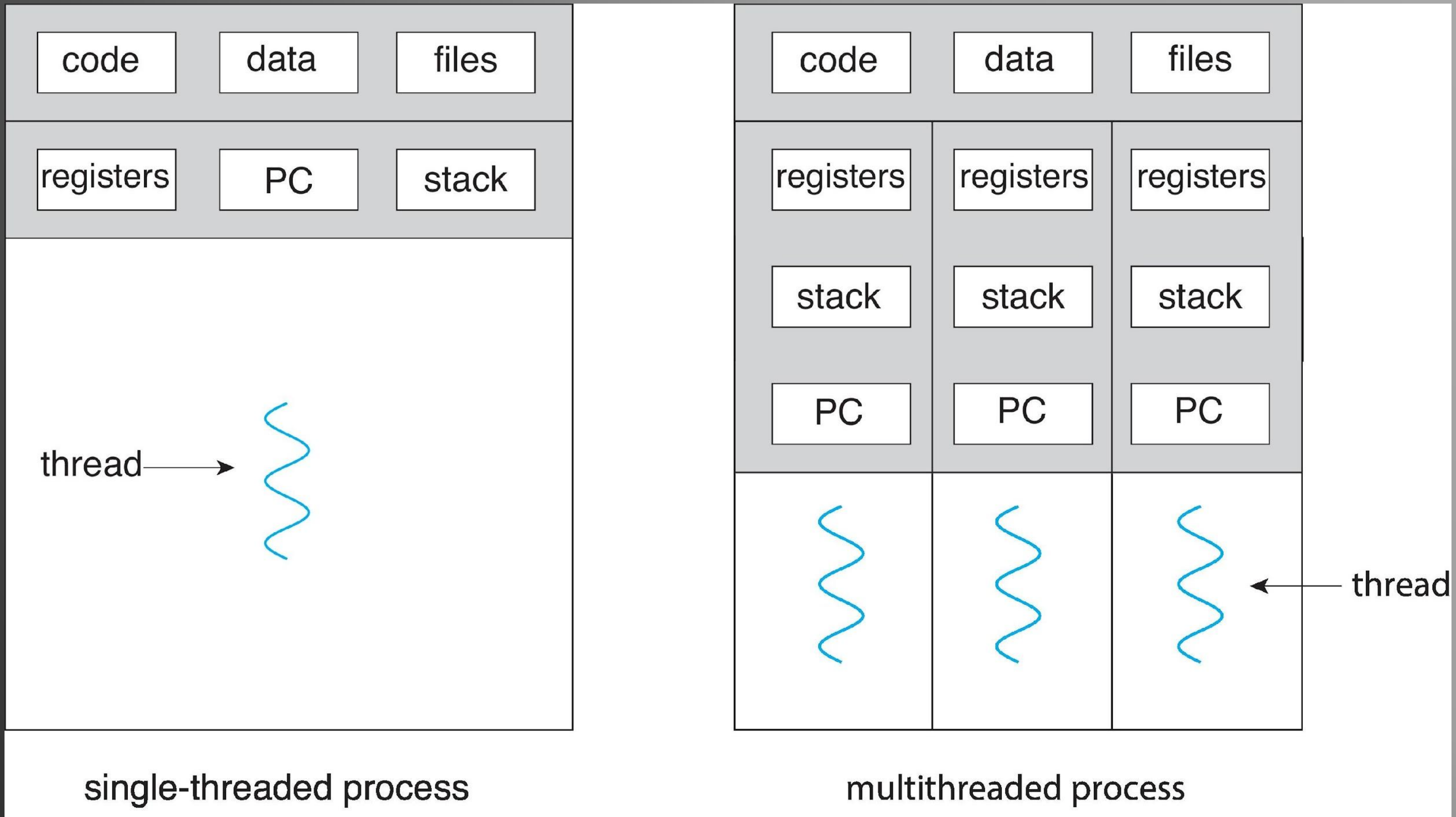
Threads share….

- Global memory
- Process ID and parent process ID
- Controlling terminal
- Process credentials (user)
- Open file information
- Timers
- ………

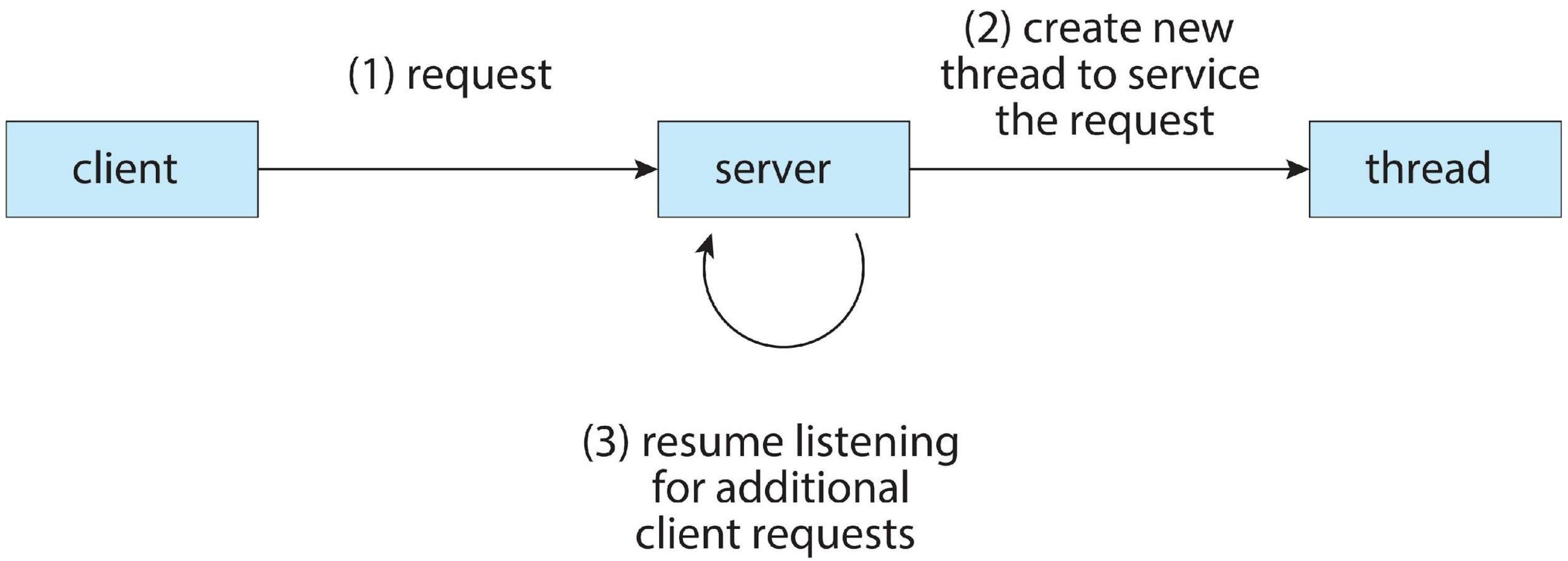
Threads specific Attributes….

- Thread ID
- Thread specific data
- CPU affinity
- Stack (local variables and function call linkage information)
- ……

Single and Multithreaded Processes



Multithreaded Server Architecture



Benefits

- **Responsiveness –** One thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations.

- **Resource Sharing –** By default threads share common code, data, and other resources, which allows multiple tasks to be performed simultaneously in a single address space.

- **Economy –** Creating and managing threads (and context switches between them) is much faster than performing the same tasks for processes.

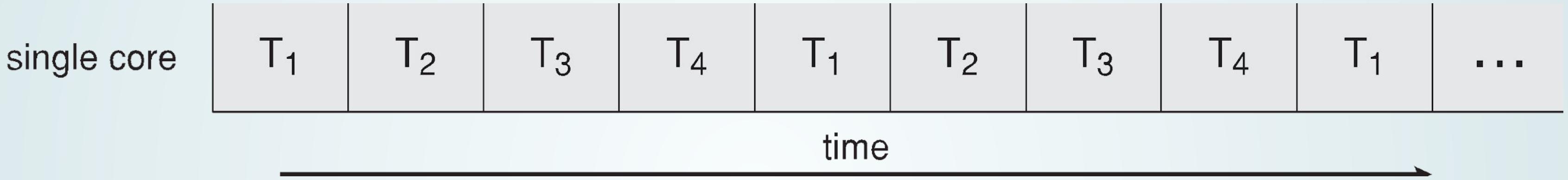
- **Scalability –** Utilization of multiprocessor architectures - A single threaded process can only run on one CPU, no matter how many may be available, whereas the execution of a multi-threaded application may be split amongst available processors.

Multicore Programming

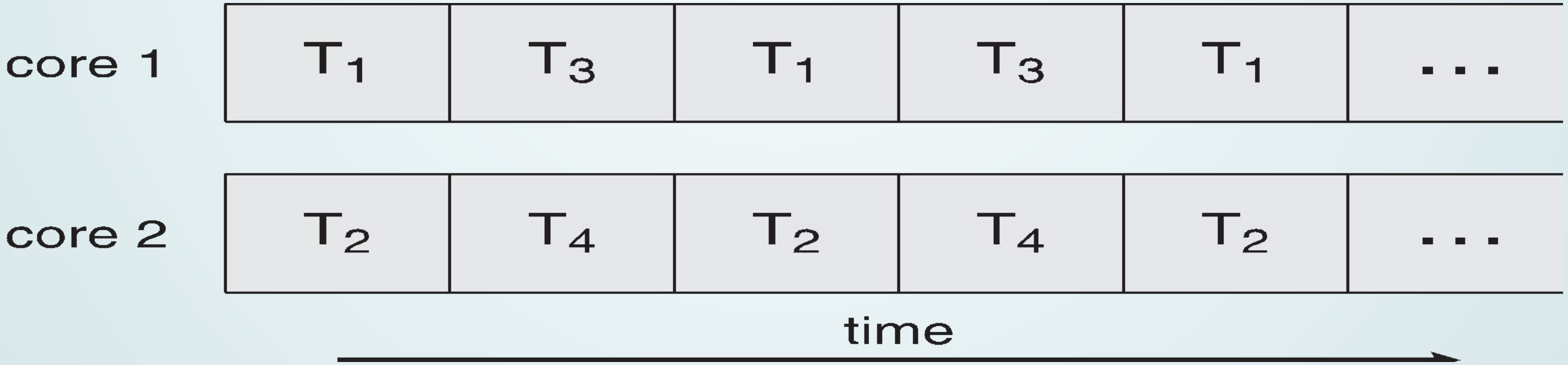
- A recent trend in computer architecture is to produce chips with multiple ***cores***, or CPUs on a single chip.
- **Parallelism** implies a system can perform more than one task simultaneously
- **Concurrency** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

Concurrency vs. Parallelism

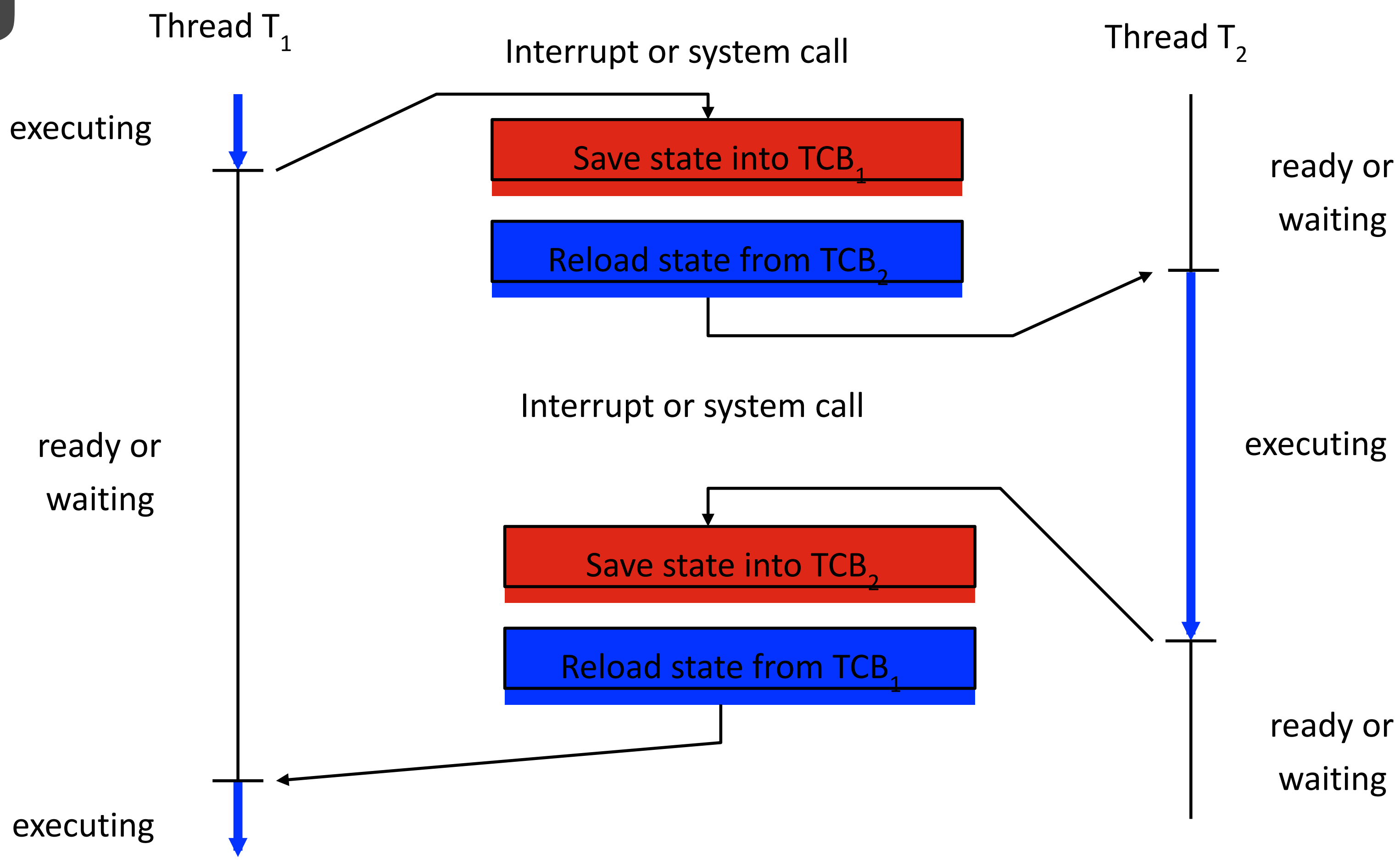
- **Concurrent execution on single-core system:**



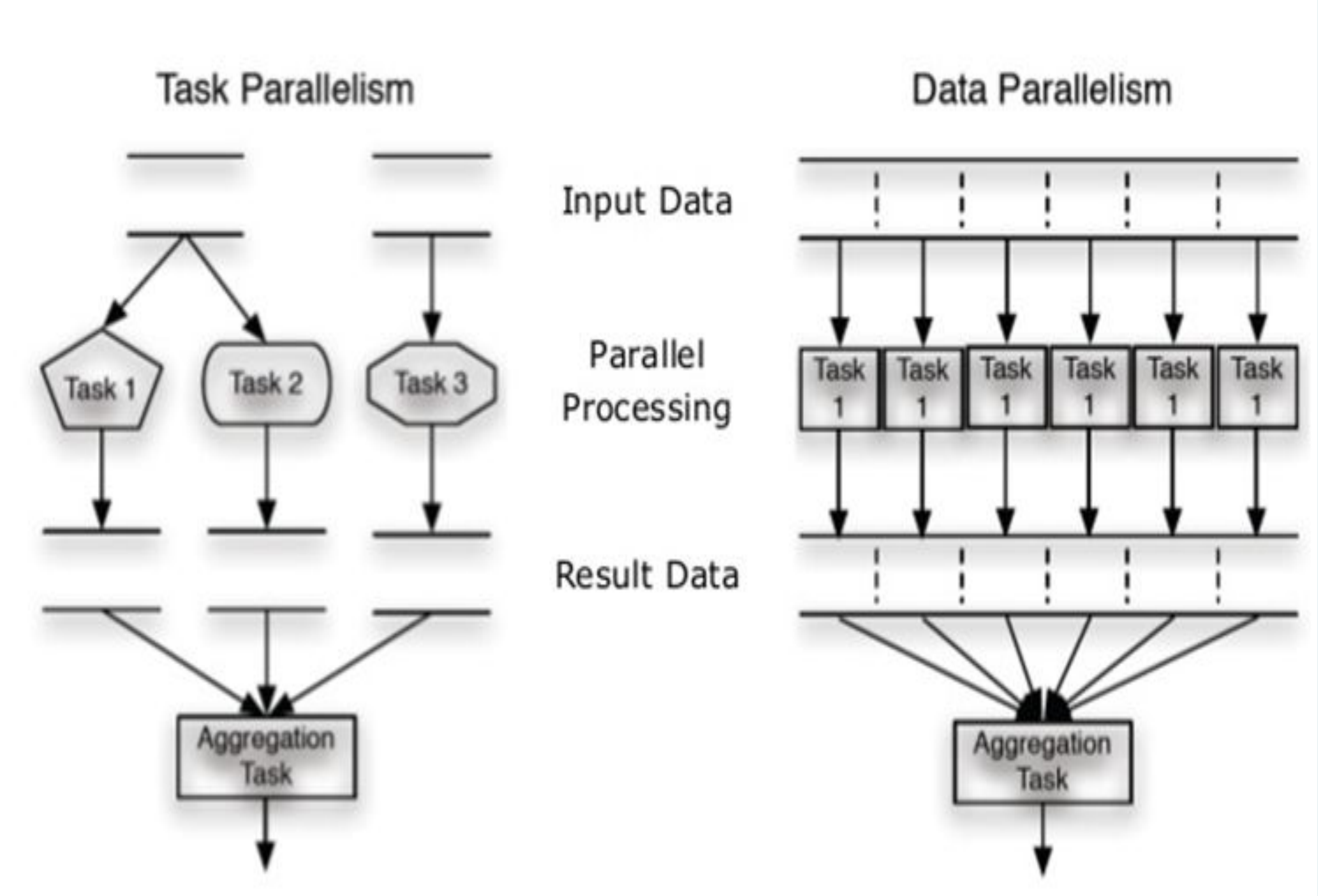
- **Parallelism on a multi-core system:**



Thread Dispatching



Types of Parallelism



Data vs. Task Parallelism

Data Parallelism	Task Parallelism
Same operations are performed on different subsets of same data.	Different operations are performed on the same or different data.
Synchronous computation	Asynchronous computation
Speedup is more as there is only one execution thread operating on all sets of data.	Speedup is less as each processor will execute a different thread or process on the same or different set of data.
Amount of parallelization is proportional to the input data size.	Amount of parallelization is proportional to the number of independent tasks to be performed
Designed for optimum load balance on multi processor system.	Load balancing depends on the availability of the hardware and scheduling algorithms like static and dynamic scheduling.

Amdahl's Law

- gives the theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved

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

Where S = portion of program executed serially
N = Processing Cores

Amdahl’s Law Example

- 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?
- S=25%=0.25, N= 2

- If we add two additional cores , calculate speedup?

Types of Threads

- Support provided at either
  - User level -> **user threads**

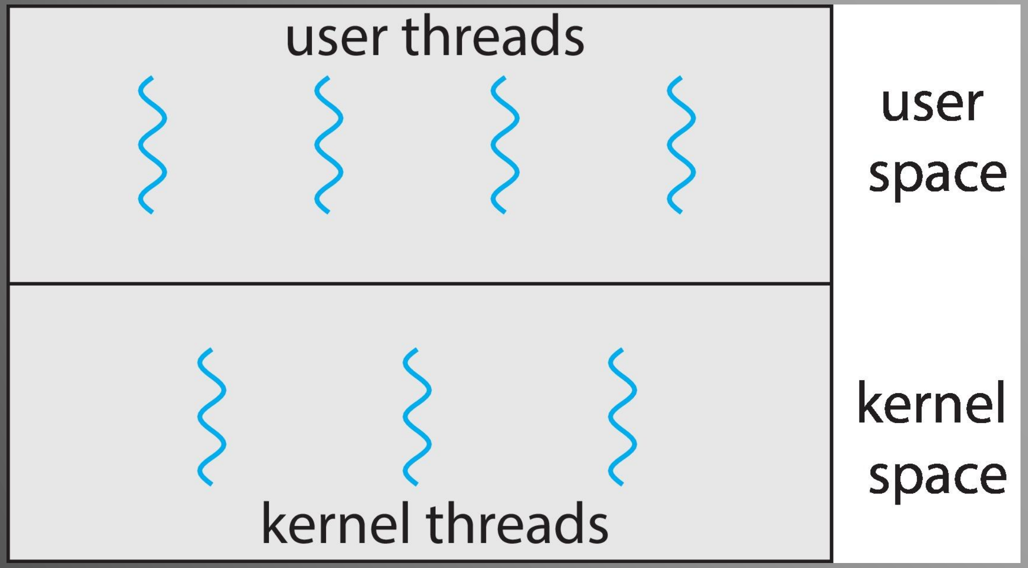
    Supported above the kernel and managed without kernel support

  - Kernel level -> **kernel threads**

    Supported and managed directly by the operating system

  What is the relationship between user and kernel threads?

User and Kernel Threads



User Threads

- Thread management done by user-level threads library

- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

Kernel Threads

- Supported by the Kernel

- Examples
  - Windows
  - Linux
  - Mac OS X
  - iOS
  - Android

Multithreading models

In a specific implementation, the user threads must be mapped to kernel threads, using these listed below Multithreading Models:

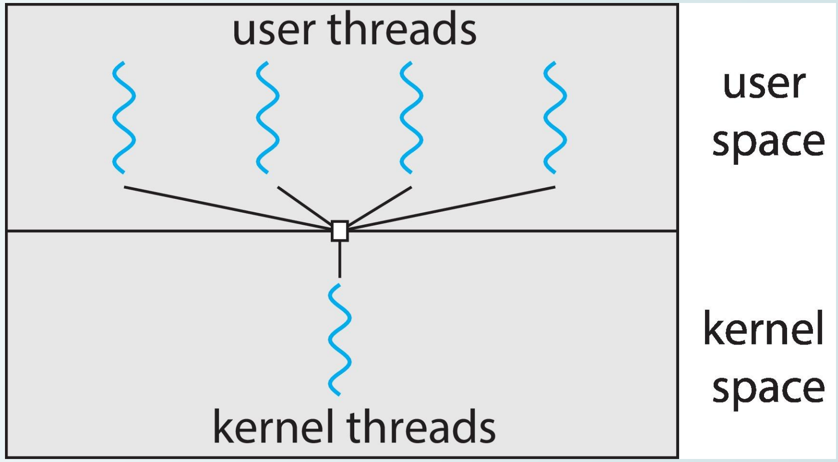
1. Many-to-One
2. One-to-One
3. Many-to-Many

User Thread – to - Kernel Thread

Many-to-One

Many user-level threads mapped to single kernel thread

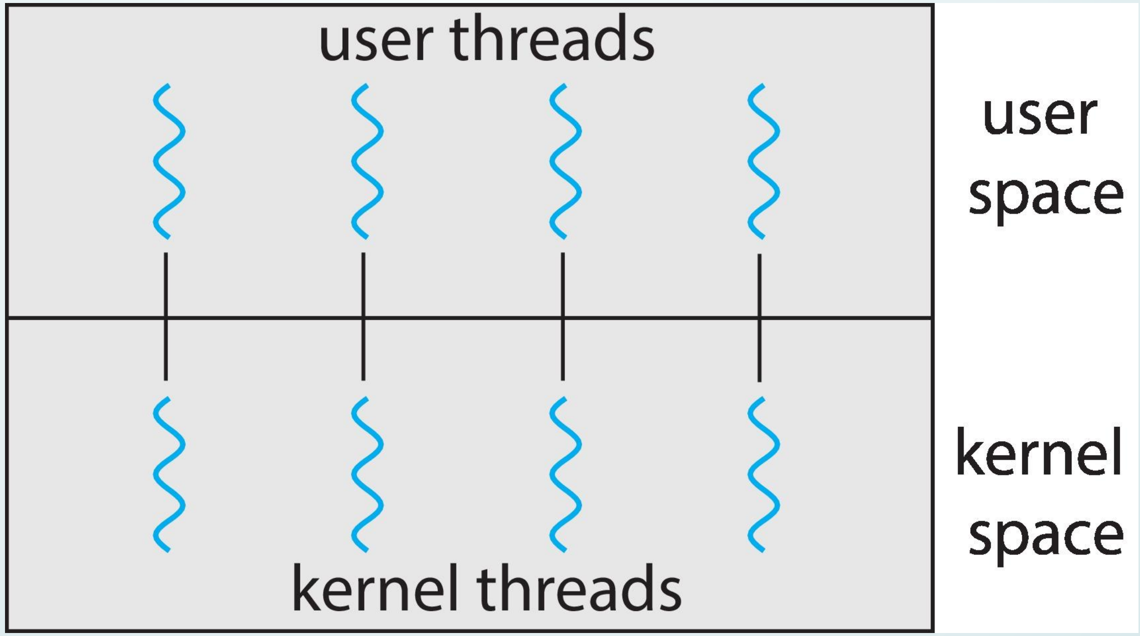
- Only one thread can access the kernel at a time,
- multiple threads are unable to run in parallel on multicore systems.
- the entire process will block if a thread makes a blocking system call



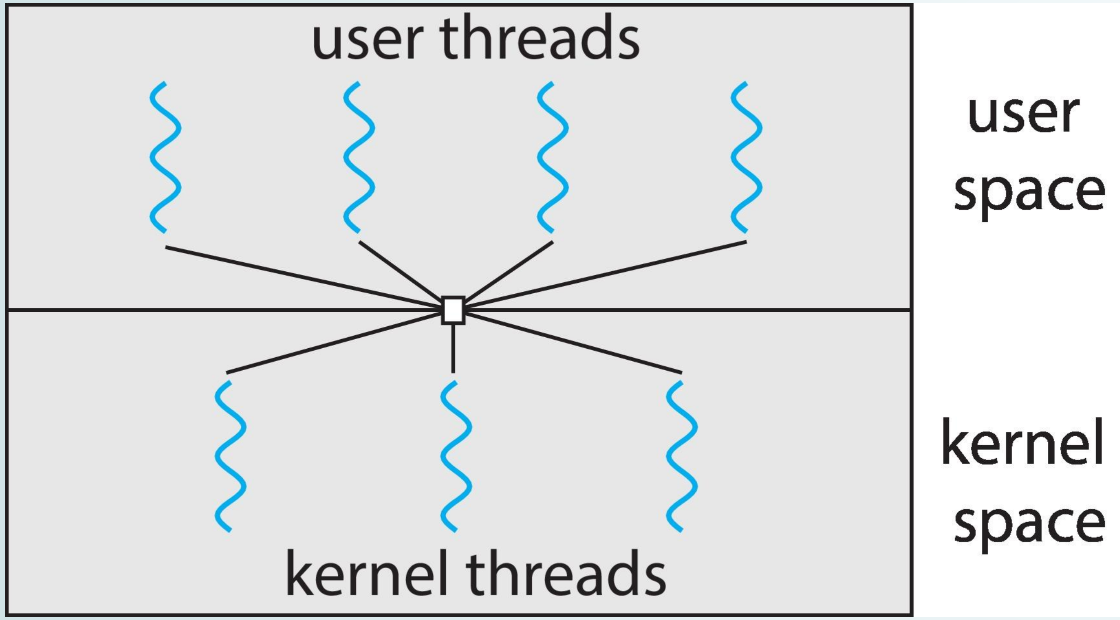
One-to-One

Each user-level thread maps to kernel thread

- more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.
- Allows multiple threads to run in parallel on multiprocessors.
- drawback is, creating a user thread requires creating the corresponding kernel thread



Many-to-Many Model



- multiplexes many user-level threads to a smaller or equal number of kernel threads
- developers can create as many user threads as necessary, and the corresponding
- kernel threads can run in parallel on a multiprocessor.
- When thread performs a blocking system call, the kernel can schedule another thread for execution.

Thread Libraries

- Three main thread libraries in use today:
  - **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
  - **Win32**
    - Kernel-level library on Windows system
  - **Java**
    - Java threads are managed by the JVM
    - Typically implemented using the threads model provided by underlying OS

POSIX Compilation on Linux

On Linux, programs that use the Pthreads API must be compiled with

-pthread or ***-lpthread***

gcc thread.c -o thread -lpthread

POSIX: Thread Creation

int pthread_create(pthread_t **thread*, const pthread_attr_t **attr*, void *(**start*)(void *), void **arg*);

- **thread* Is the location where the ID of the newly created thread should be stored, or NULL if the thread ID is not required.
- *attr* Is the thread attribute object specifying the attributes for the thread that is being created. If *attr* is NULL, the thread is created with default attributes.
- *start* Is the main function for the thread; the thread begins executing user code at this address.
- *arg* Is the argument passed to *start*.

POSIX: Thread ID

```
#include <pthread.h>

pthread_t pthread_self()
```

returns : ID of current (this) thread

POSIX: Wait for Thread Completion

```
#include <pthread.h>

pthread_join (thread, NULL)
```

returns : 0 on success, some error code on failure.

POSIX: Thread Termination

```
#include <pthread.h>

Void pthread_exit (return_value)
```

Threads terminate in one of the following ways:

1. The thread's start functions performs a return specifying a return value for the thread.
2. Thread receives a request asking it to terminate using pthread_cancel()
3. Thread initiates termination pthread_exit()
4. Main process terminates

```
int main()
{
    pthread_t thread1, thread2;  /* thread variables */
    thdata data1, data2;         /* structs to be passed to threads */

    /* initialize data to pass to thread 1 */
    data1.thread_no = 1;
    strcpy(data1.message, "Hello!");

    /* initialize data to pass to thread 2 */
    data2.thread_no = 2;
    strcpy(data2.message, "Hi!");

    /* create threads 1 and 2 */
    pthread_create (&thread1, NULL, (void *) &print_message_function, (void *) &data1);
    pthread_create (&thread2, NULL, (void *) &print_message_function, (void *) &data2);

    /* Main block now waits for both threads to terminate, before it exits
       If main block exits, both threads exit, even if the threads have not
       finished their work */
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

exit(0);
}
```

Example code but not complete

Implicit Threading

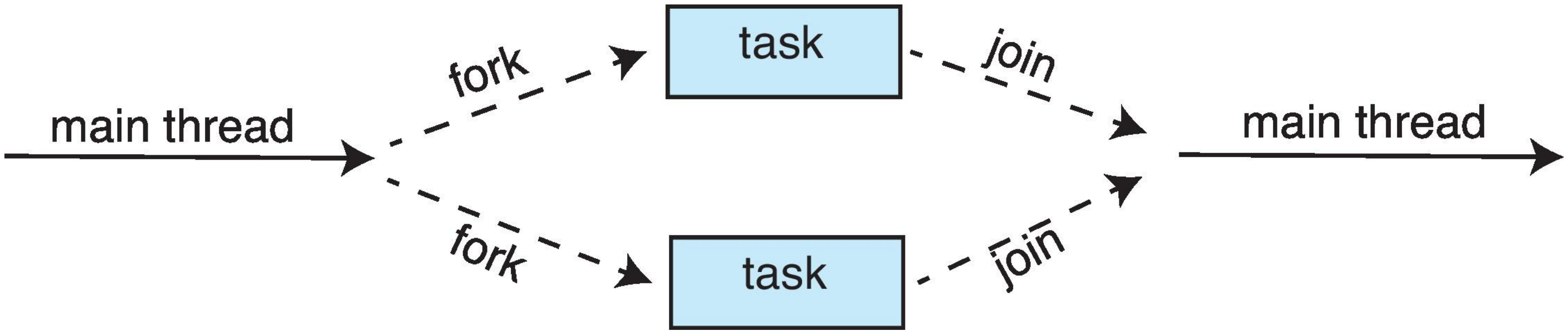
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- These listed below methods explored
  1. Thread Pools
  2. Fork Join
  3. OpenMP

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

Fork-Join Parallelism

- Multiple threads (tasks) are **forked**, and then **joined**.



OpenMP

- An Application Program Interface (API) that may be used to explicitly direct multithreaded, shared memory parallelism

- Three main API components
  - Compiler directives
  - Runtime library routines
  - Environment variables

- Portable & Standardized
  - API exist both C/C++ and Fortan 90/77
  - Multi platform Support (Unix, Linux etc.)

OpenMP Compilation

- GCC

```
bash: $ gcc  -fopenmp  hi-omp.c  -o   hi-omp.x
```

OpenMP Directives

```
#pragma omp parallel default(shared) private(beta,pi)
```

#pragma omp barrier

Each thread waits at the barrier until all threads have reached it.

#pragma omp for

Distributes the iterations of a loop over multiple threads

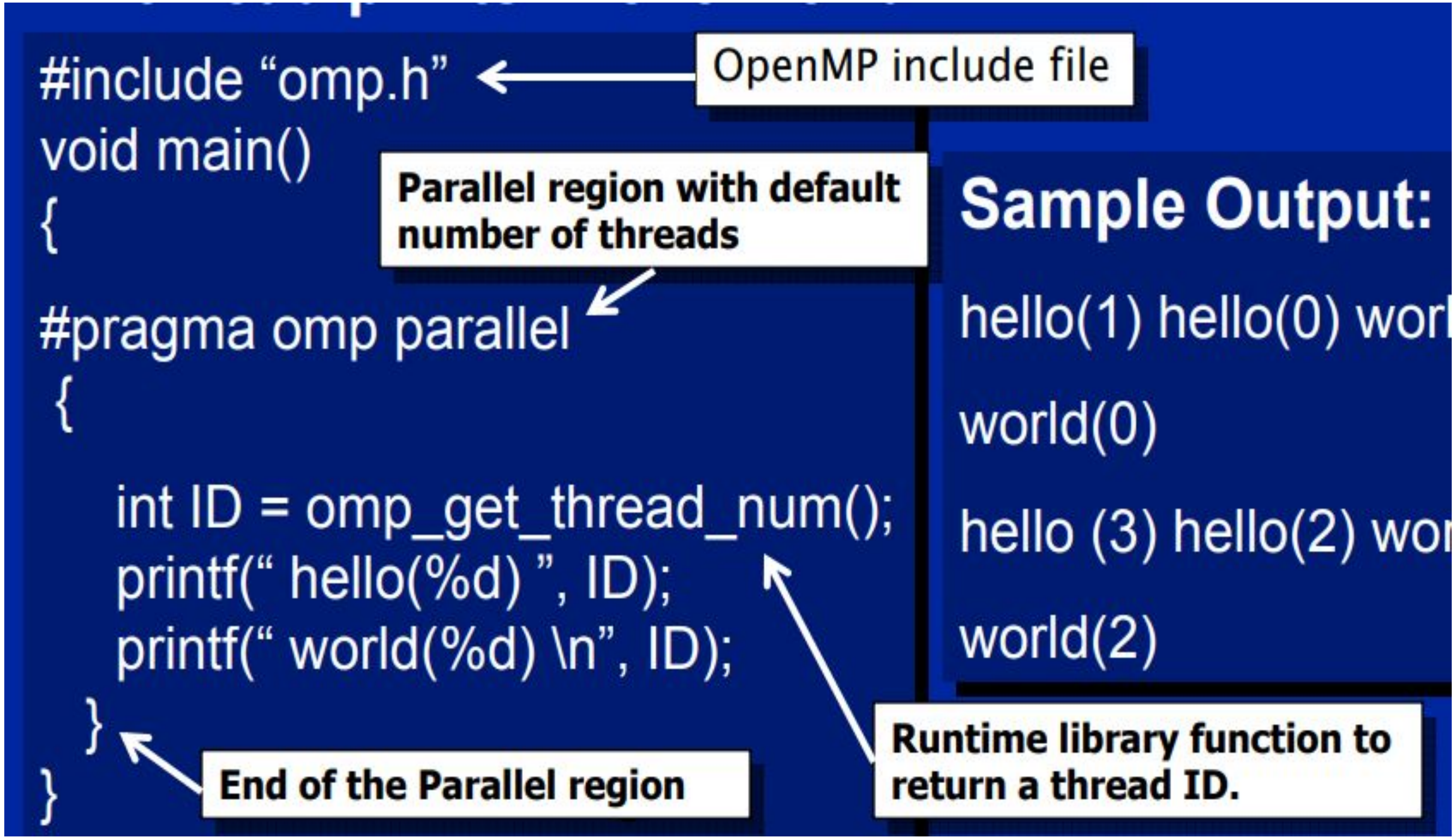
OpenMP threads

- **Thread Creation:**
- omp_get_num_threads()

Returns number of threads in parallel region Returns 1 if called outside parallel region

- **Thread Id:**
- omp_get_thread_num()
- Returns id of thread in team Value between [0,n-1] // where n = #threads Master thread always has id 0

Open MP Example



Thread-Local Storage

- **Thread-local storage** (**TLS**) allows each thread to have its own copy of data.
- major thread libraries (pThreads, Win32, Java) provide support for thread-specific data, known as **thread-local storage** or **TLS.**

Course Supervisor: ANAUM HAMID

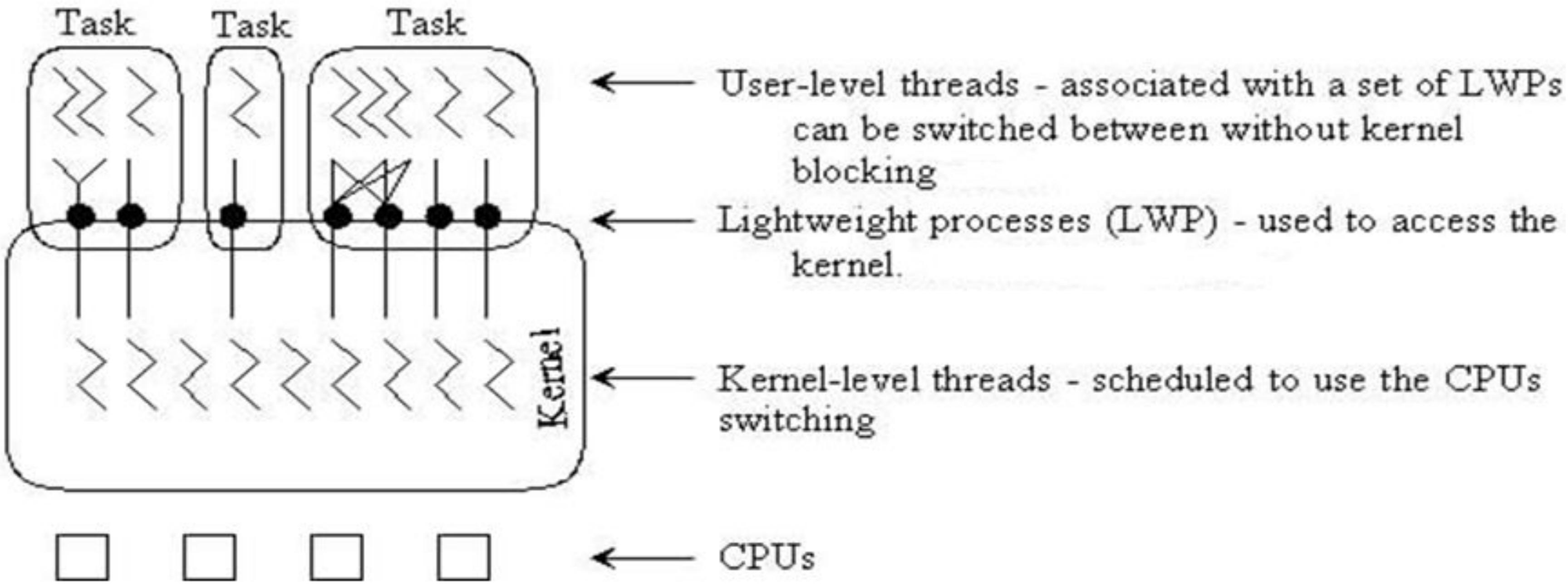
Signal Handling

- **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
- Where should a signal be delivered for multi-threaded?
  1. Deliver the signal to the thread to which the signal applies
  2. Deliver the signal to every thread in the process
  3. Deliver the signal to certain threads in the process
  4. Assign a specific thread to receive all signals for the process

THREAD SCHEDULING

- In systems that support user and kernel-level threads, kernel-level threads are scheduled by the OS.
- Kernel-level threads instead of processes are scheduled.
- User-level threads are managed by a thread library.
- To run on the CPU, the user-level thread must be mapped on an associated kernel-level thread

User vs. Kernel Thread



THANK YOU