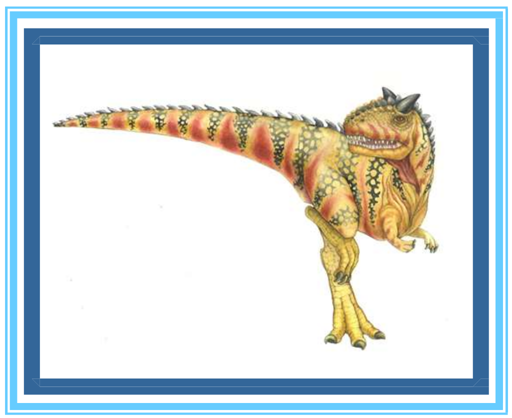
Chapter 4: Threads & Concurrency

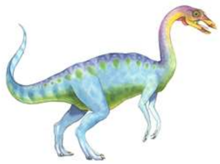


Outline

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples

Objectives

- Identify the basic components of a thread, and contrast threads and processes
- Describe the benefits and challenges of designng multithreaded applications
- Illustrate different approaches to implicit threading including thread pools, fork-join, and Grand Central Dispatch
- Describe how the Windows and Linux operating systems represent threads
- Design multithreaded applications using the Pthreads, Java, and Windows threading APIs



Terminology

- Multiprogramming
  - A computer running more than one program at a time (like running Excel and Firefox simultaneously)
  - Context switching
- Multiprocessing
  - A computer using more than one CPU (processor) or core at a time
- Multitasking
  - Multitasking is a logical extension of multi programming (time sharing)
  - Tasks sharing a common resource (like 1 CPU)
- Multithreading
  - Thread (a code segments)
  - is an extension of multitasking

https://www.geeksforgeeks.org/difference-between-multitasking-multithreading-and-multiprocessing/

https://www.8bitavenue.com/difference-between-multiprogramming-multitasking-multithreading-and-multiprocessing/

Two Characteristics of Processes

Resource Ownership

Process includes a virtual address space to hold the process image

- The OS performs a protection function to prevent unwanted interference between processes with respect to resources

Scheduling/Execution

Follows an execution path that may be interleaved with other processes

- A process has an execution state (Running, Ready, etc.) and a dispatching priority, and is the entity that is scheduled and dispatched by the OS

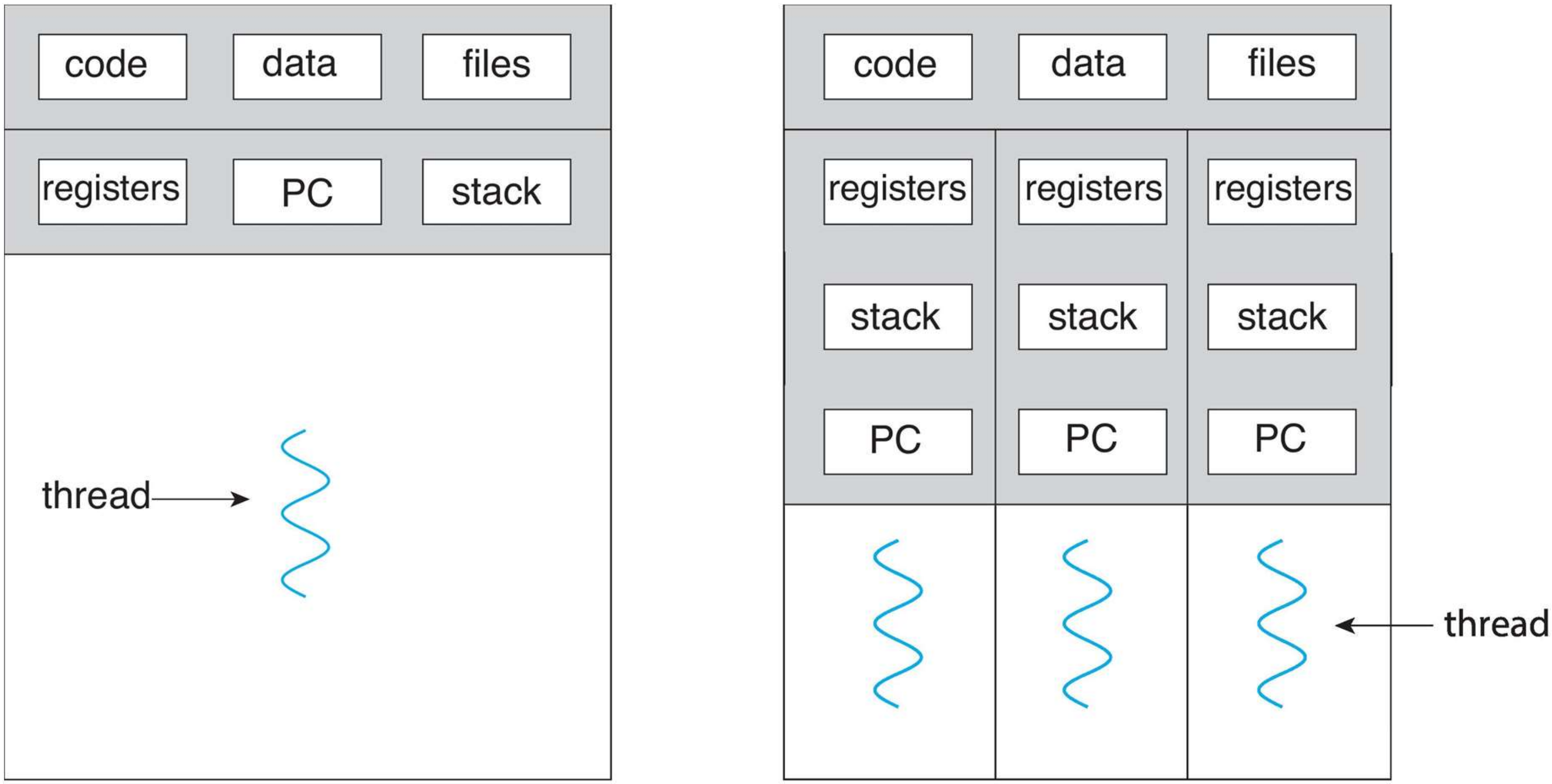
Processes and Threads

- These two process characteristics are treated independently by the operating system
  - The unit of execution (CPU utilization) is referred to as a ***thread*** or lightweight process.
  - The unit of resource ownership is referred to as a ***process*** or task.

Processes and Threads

- The unit of dispatching is referred to as a ***thread*** or ***lightweight process***
- The unit of resource ownership is referred to as a ***process*** or ***task***
- ***Multithreading*** **-** The ability of an OS to support multiple, concurrent paths of execution within a single process

Single and Multithreaded Processes



single-threaded process

multithreaded process

Motivation

- Most modern applications are multithreaded
- Example:
  - Web browser: one thread displays images or text while another thread receives data from the network
  - Word processor: a thread for displaying the graphics, another one for responding to keystrokes, and a third thread for performing spelling and grammar checking.
- 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

Key Benefits of Threads

- Takes less time to create a new thread than a process
- Less time to terminate a thread than a process
- Switching between two threads takes less time than switching between processes
- Threads enhance efficiency in communication between programs

Threads can communicate with each other without invoking the kernel.

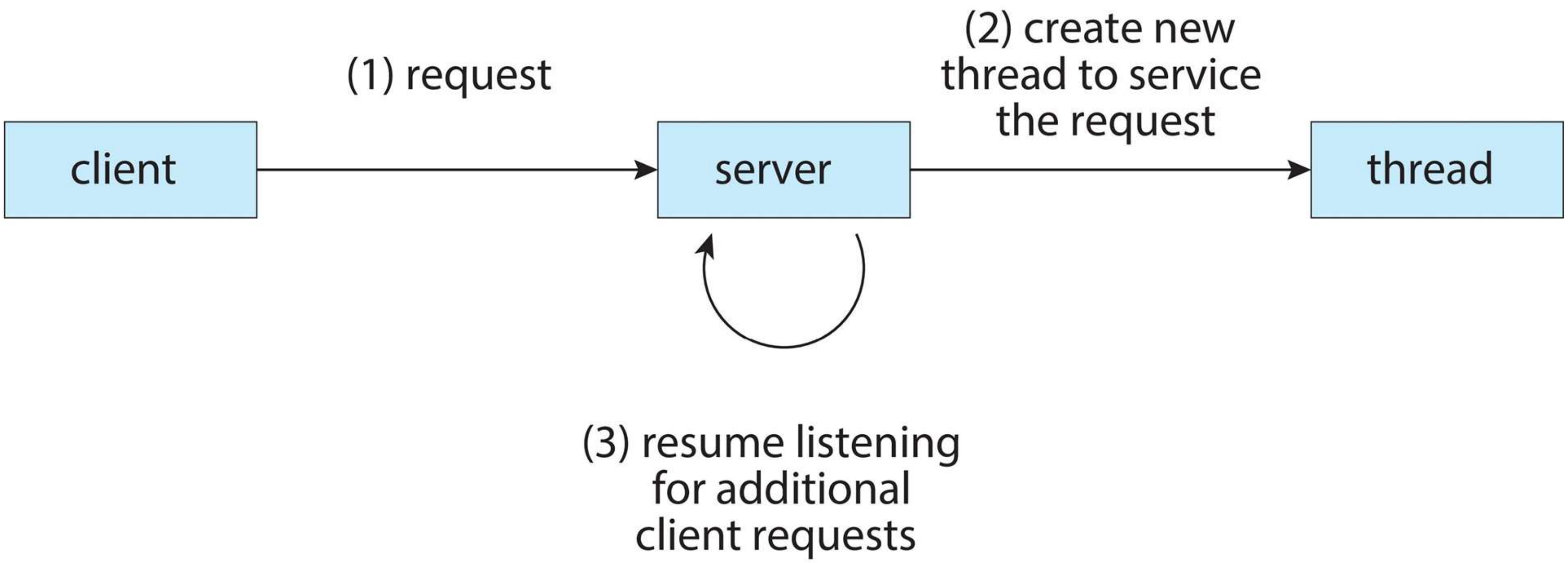
Threads

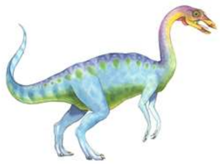
- In an OS that supports threads, scheduling and dispatching is done on a **thread basis**

Most of the state information dealing with execution is maintained in **thread-level** data structures

- Suspending a process involves suspending all threads of the process
- Termination of a process terminates all threads within the process

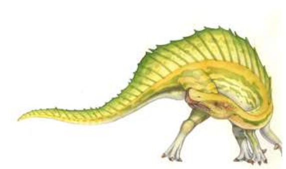
Multithreaded Server Architecture





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
  - Utilization of multiprocessor architectures. Such that threads can run in parallel on different processors.



Single Threaded Approaches

- A single thread of execution per process, in which the concept of a thread is not recognized, is referred to as a single-threaded approach
- MS-DOS is an example

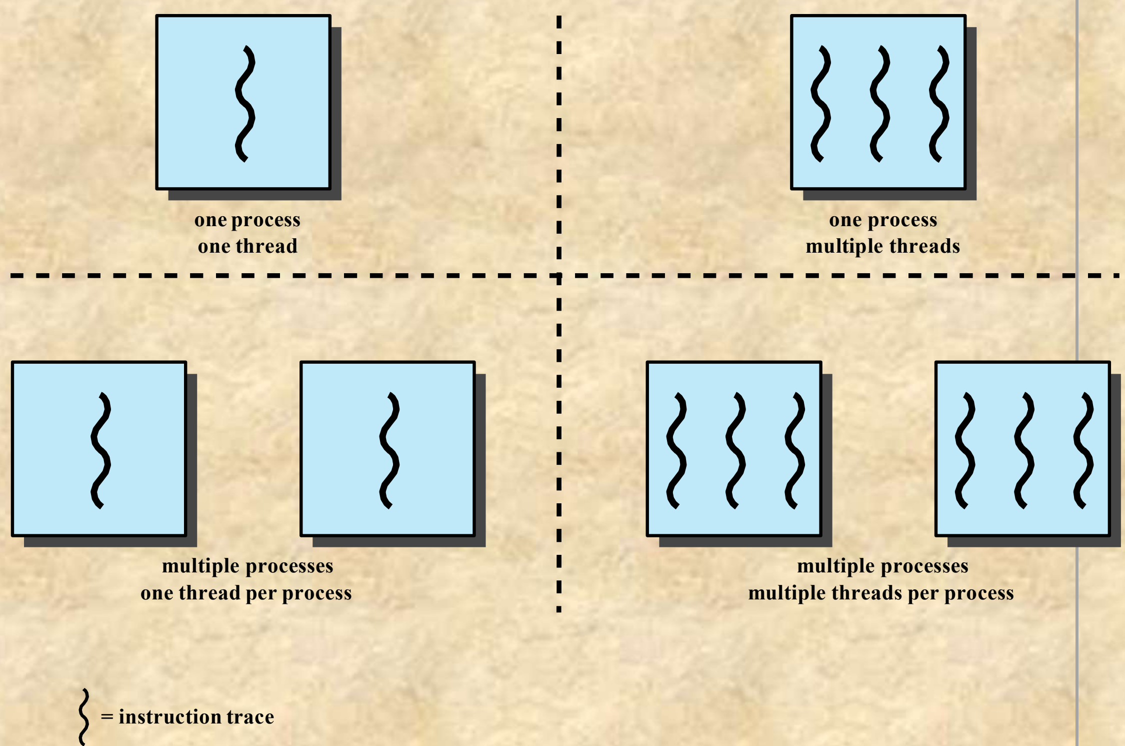


Figure 4.1 Threads and Processes

Multithreaded Approaches

- The right half of Figure 4.1 depicts multithreaded approaches
- A Java run-time environment is an example of a system of one process with multiple threads

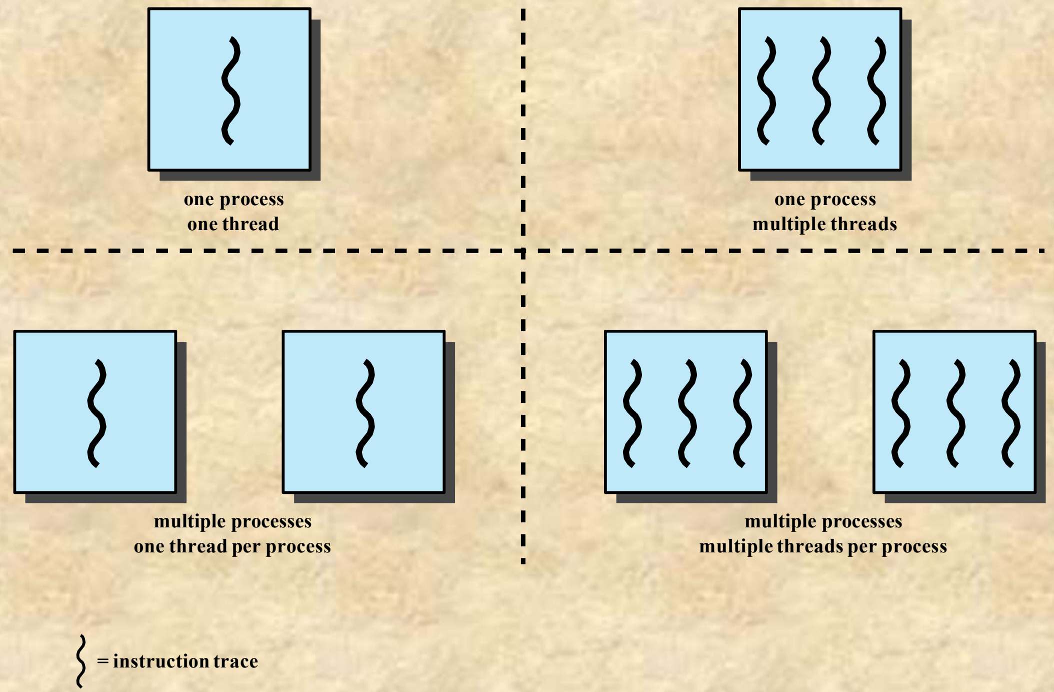


Figure 4.1 Threads and Processes

One or More Threads in a Process

Each thread has:

- An execution state (Running, Ready, etc.)
- A 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 processes, shared with all other threads in that process

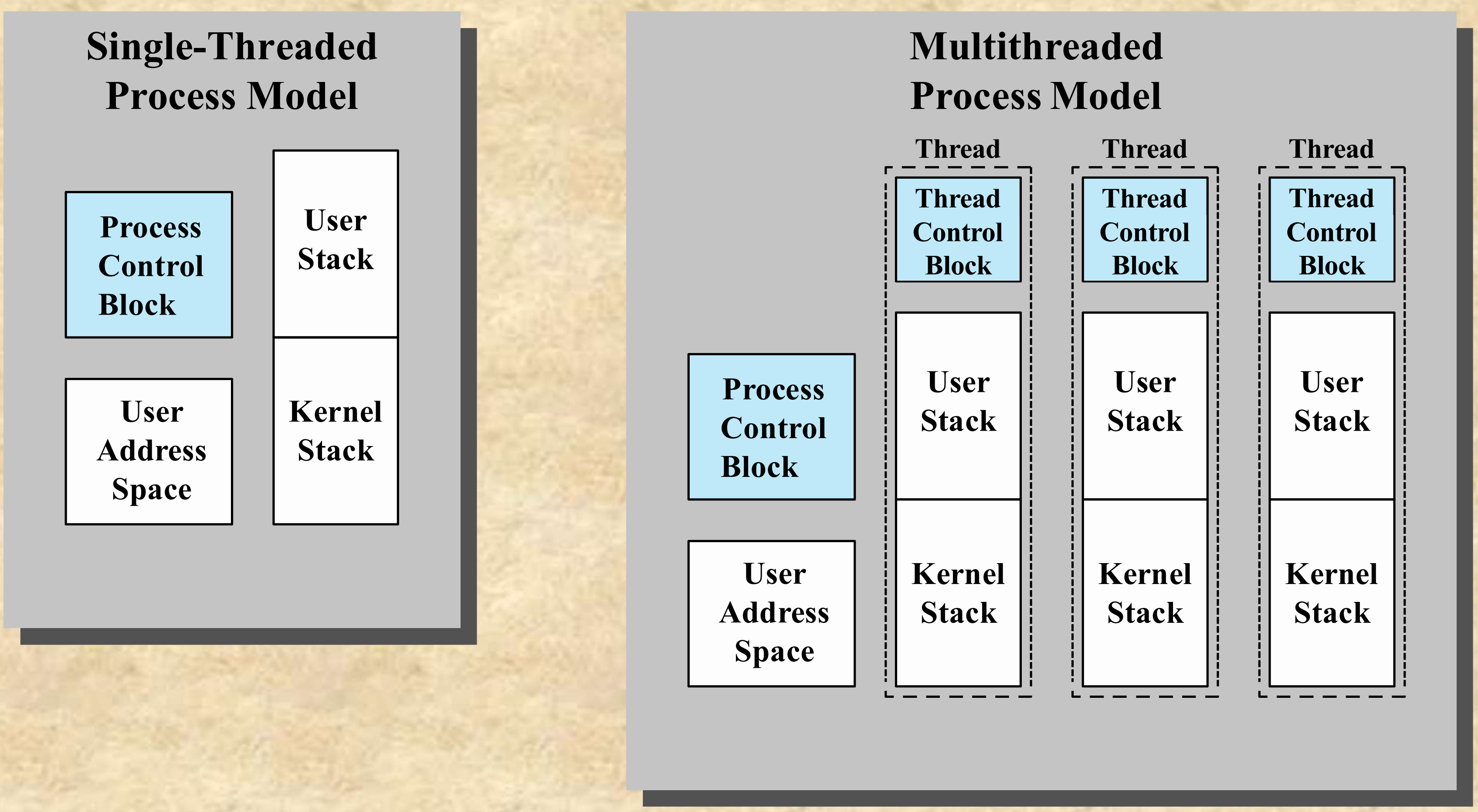
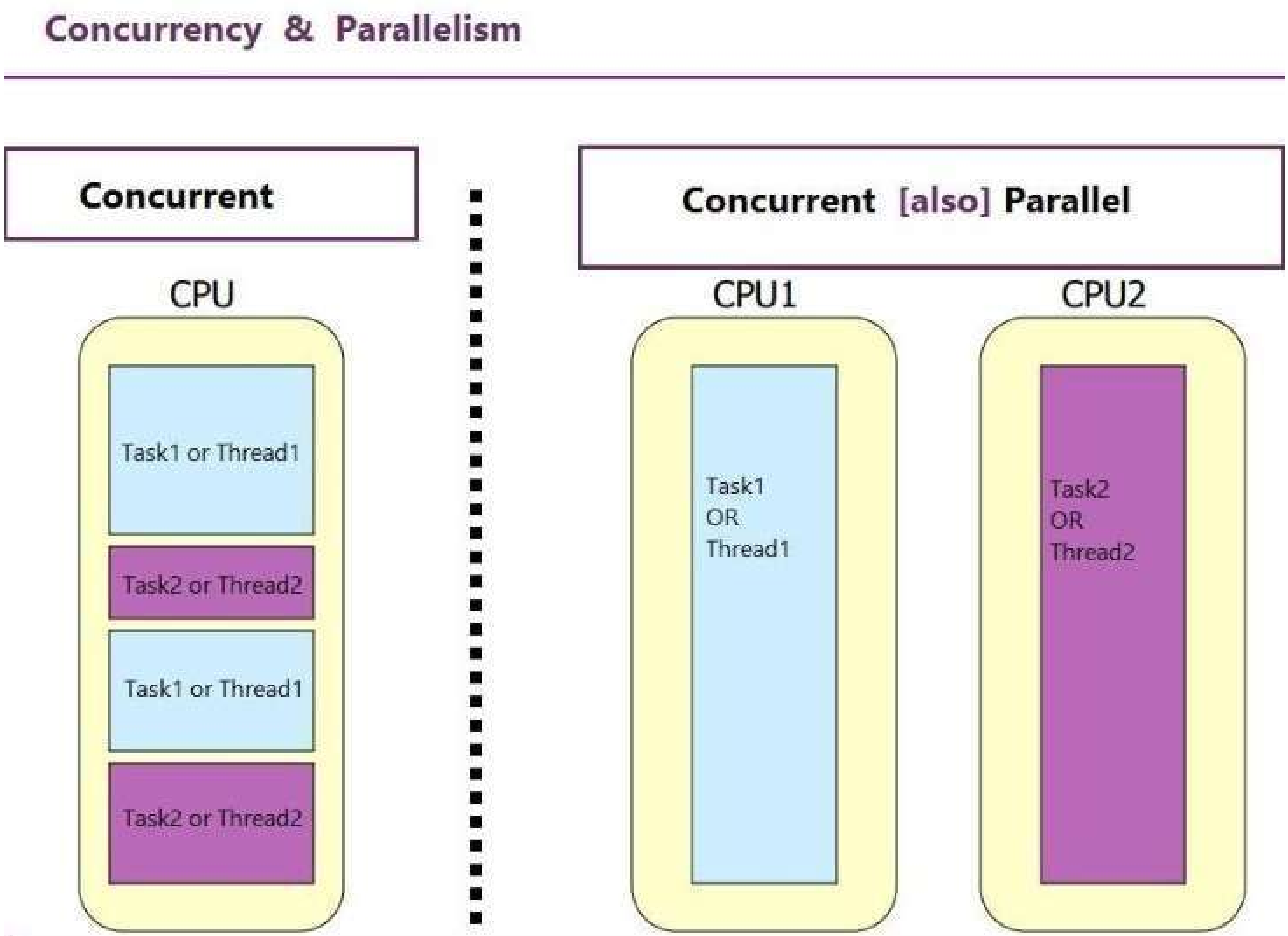


Figure 4.2 Single Threaded and Multithreaded Process Models

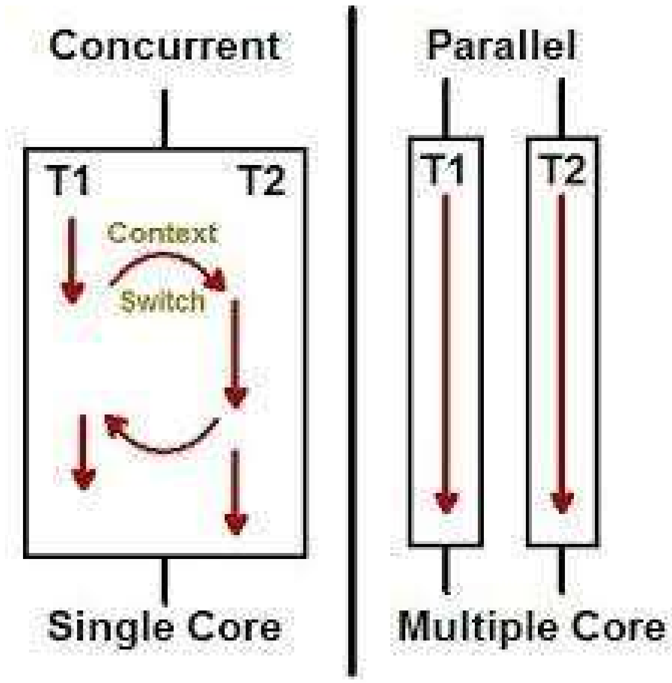
Multicore Programming

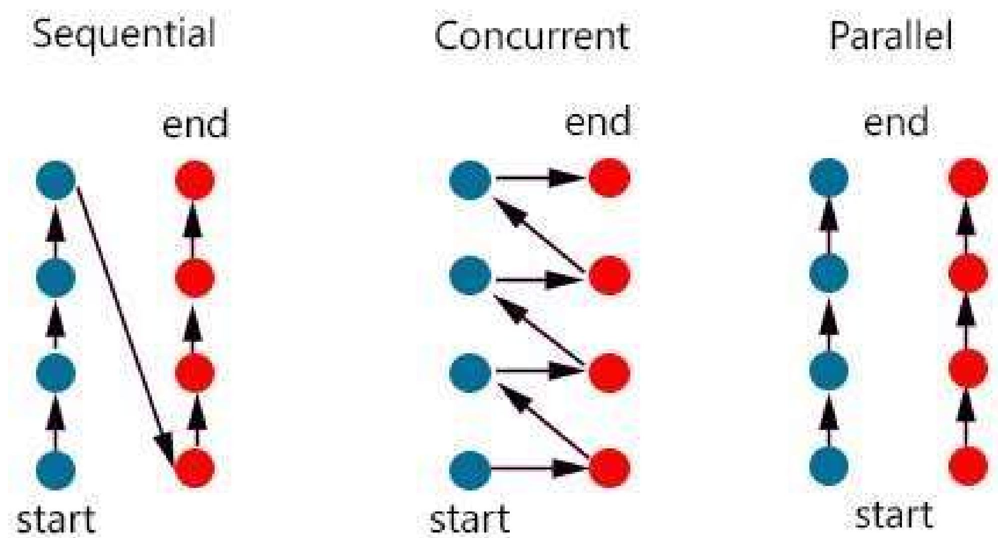
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously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

Concurrency vs. Parallelism



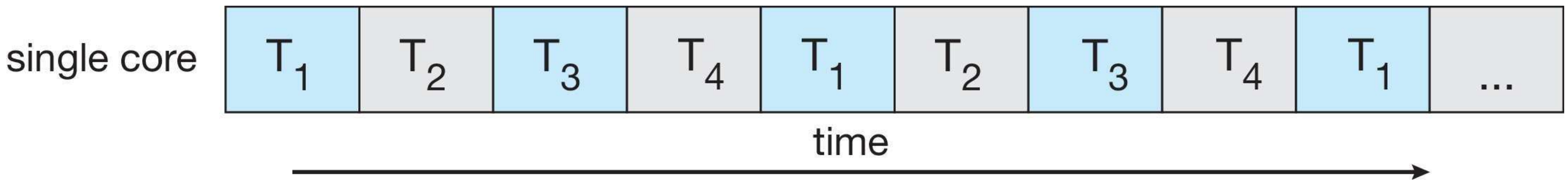
Concurrency vs. Parallelism



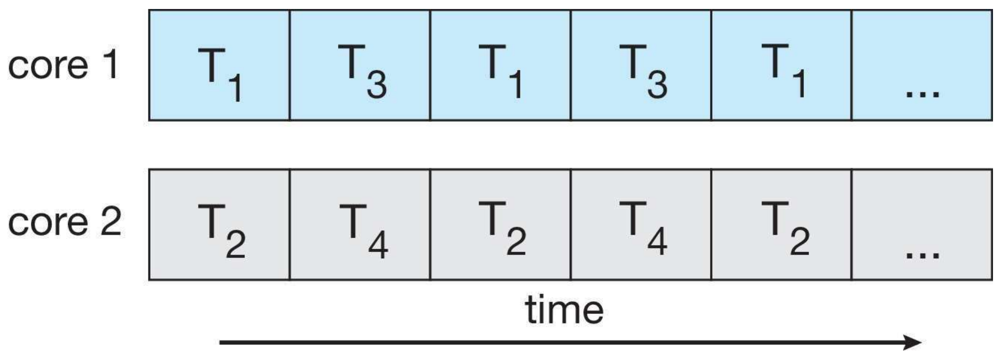


Concurrency vs. Parallelism

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



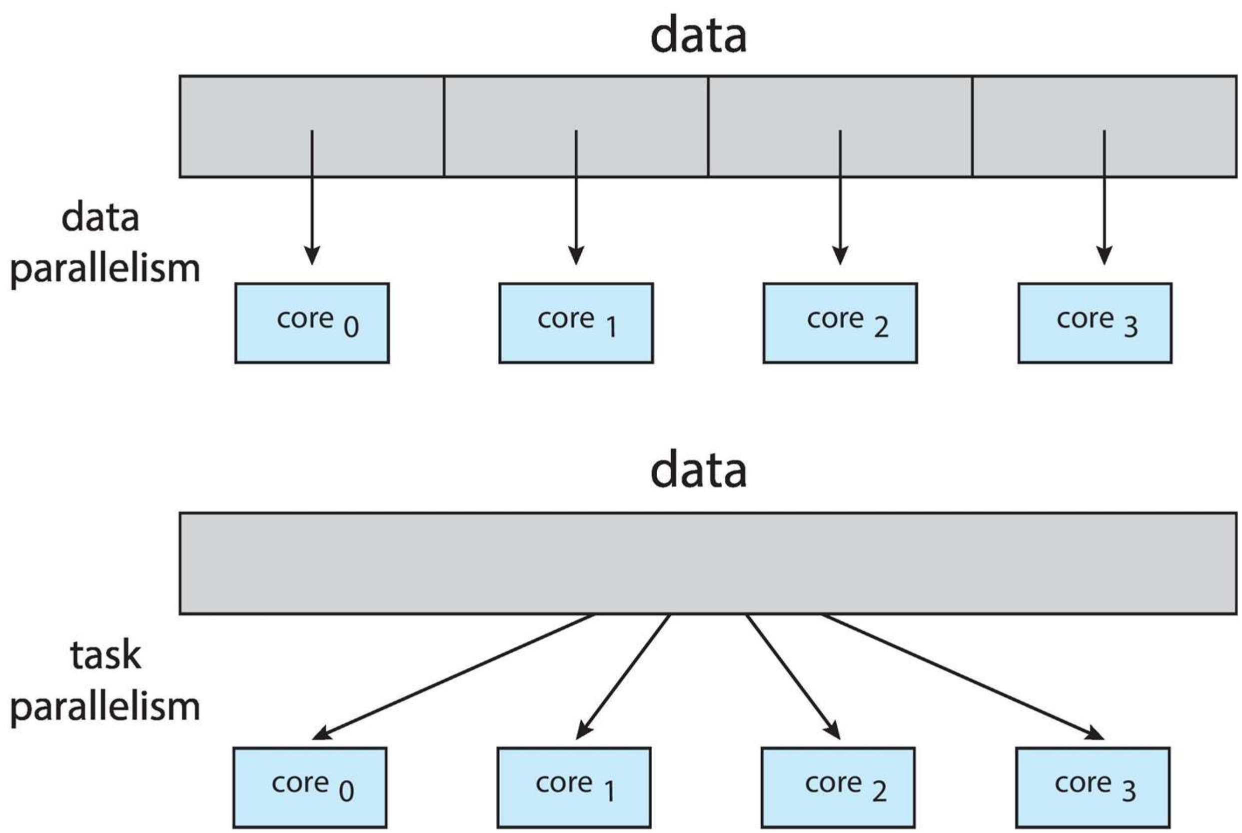
- **Parallelism on a multi-core system:**



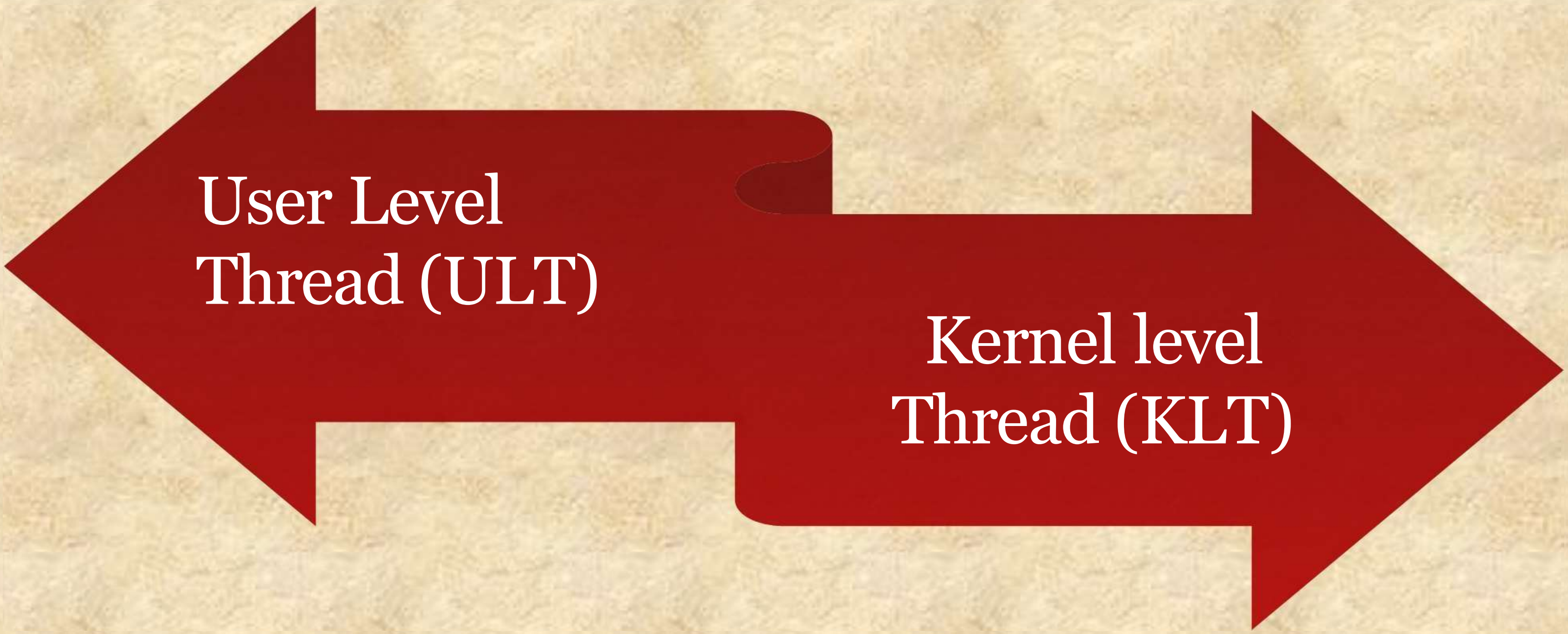
Multicore Programming

- 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

Data and Task Parallelism



Types of Threads



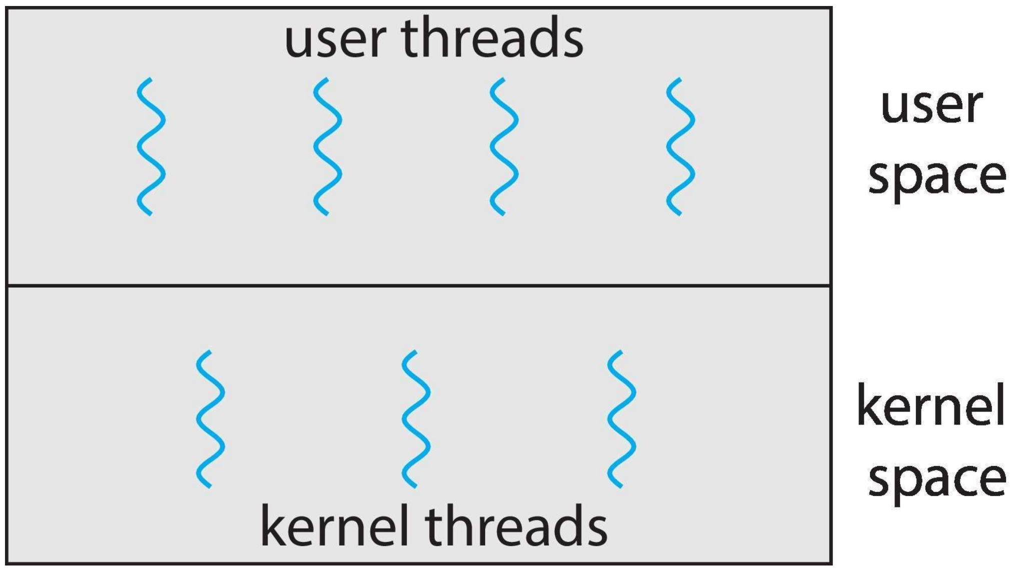
Threads Implementation

- **User Level Thread (ULT)**
- **Kernel Level Thread (KLT)** also called:
  - Kernel-supported thread
  - Lightweight process

User Threads and Kernel Threads

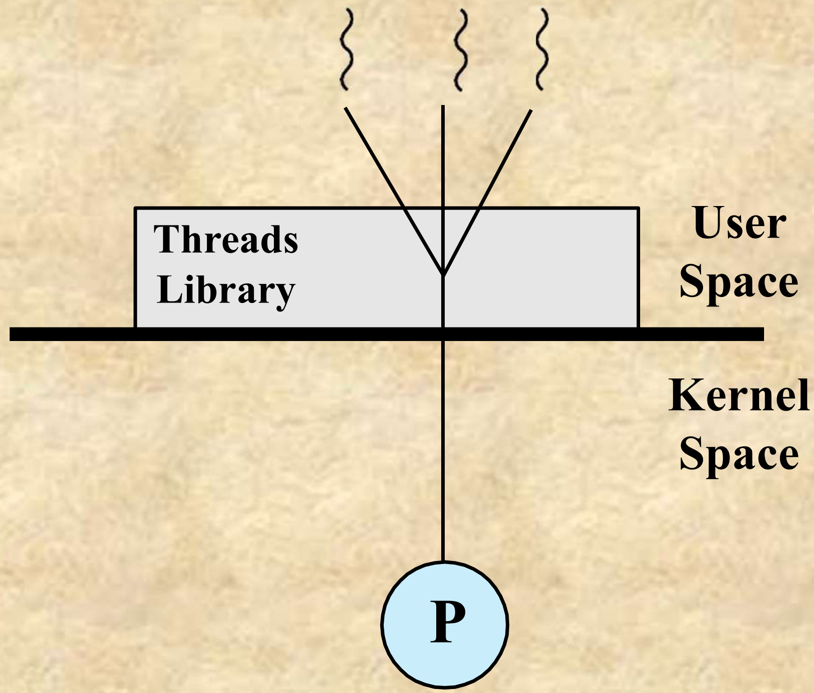
- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general-purpose operating systems, including:
  - Windows
  - Linux
  - Mac OS X
  - iOS
  - Android

User and Kernel Threads



User-Level Threads (ULTs)

- All thread management is done by the application using a thread library
  - The **user library** contains code for creating threads, destroying threads, scheduling thread execution and …
- The kernel is not aware of the existence of threads



(a) Pure user-level

Advantages of ULTs

Scheduling can be application specific and specify by programmer

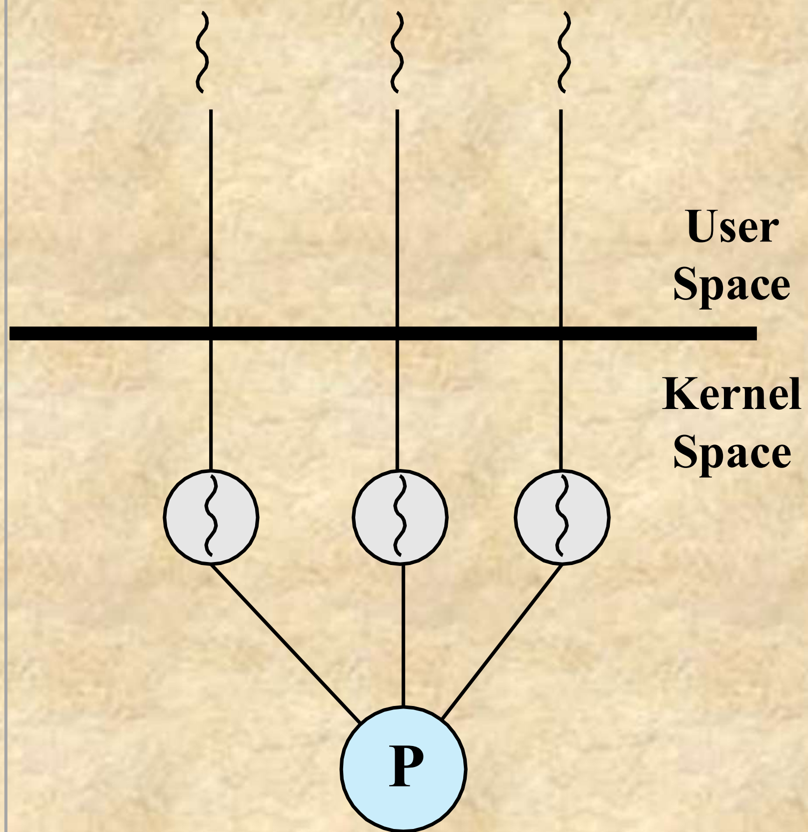
ULTs can run on any OS, even the ones that do not support multithreading like embedded OSs.

Less overhead: Thread switching does not require kernel mode privileges

Disadvantages of ULTs

- In a typical OS many system calls are blocking
  - As a result, when a ULT executes a system call, not only is that thread blocked, but all of the threads within the process are blocked as well
- In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing
  - A kernel assigns one process to only one processor at a time, therefore, only a single thread within a process can execute at a time

Kernel-Level Threads (KLTs)



(b) Pure kernel-level

- Thread management is done by the **kernel**
  - There is no thread management code in the application level, simply an application programming interface (API) to the kernel thread facility
  - The kernel maintains context information for the process and threads
  - Scheduling is done on a thread basis
  - Windows is an example of this approach

Advantages of KLTs

- The kernel can simultaneously schedule multiple threads from the same process on multiple processors
- If one thread in a process is blocked, the kernel can schedule another thread of the same process
- Kernel routines themselves can be multithreaded (in all modern OSs)

Disadvantage of KLTs

- The transfer of control from one thread to another within the same process requires a mode switch to the kernel (increase **overhead** for OS)

Operation	User-Level Threads	Kernel -Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

Table 4.1
Thread and Process Operation Latencies

Combined Approaches

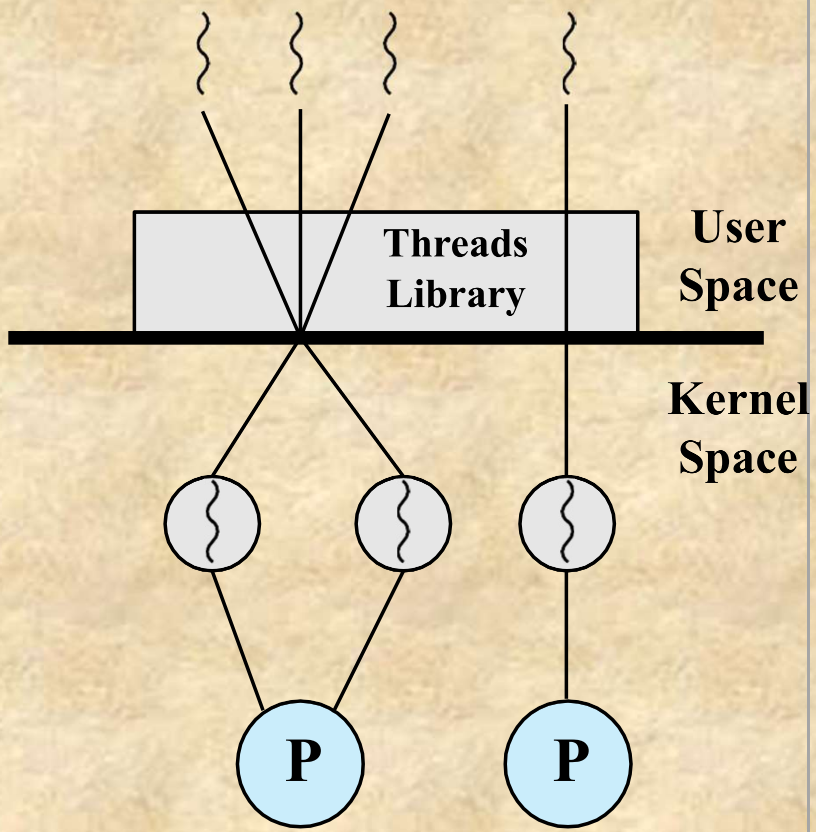
- Thread creation is done completely in the user space, as is the bulk of the scheduling and synchronization of threads within an application

- The multiple ULTs from a single application are mapped onto some (smaller or equal) number of KLTs.

- **Solaris** is a good example

  - Windows and Linux are Kernel-level

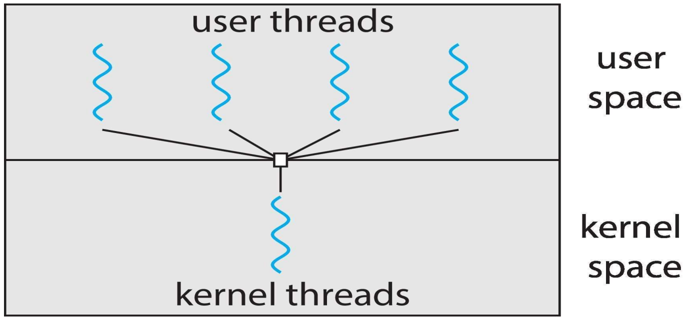
- JVM: mapped user Java threads into Kernel threads. Possibilities: One-to-one, many-to-many and …The mapping can be different in Windows from Linux and …



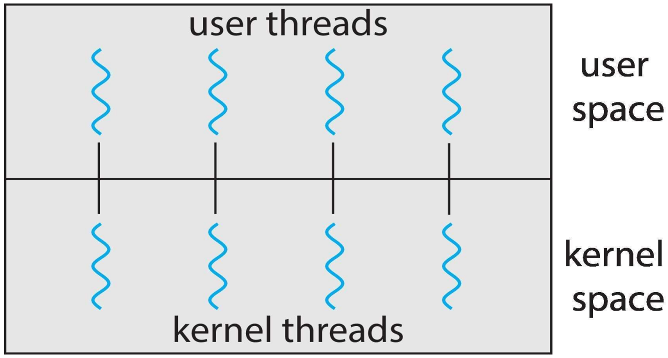
(c) Combined

Multithreading Models

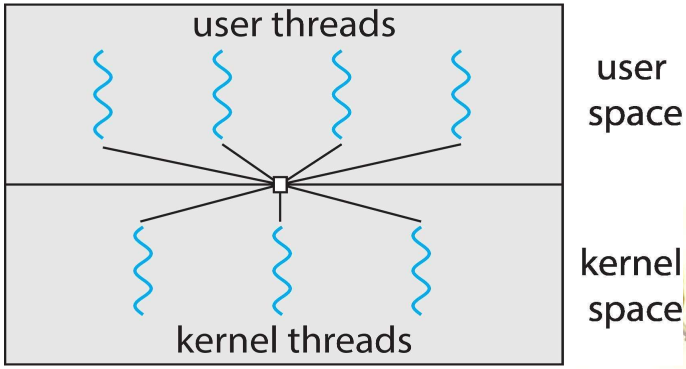
- Many-to-One == ULT



- One-to-One == KLT

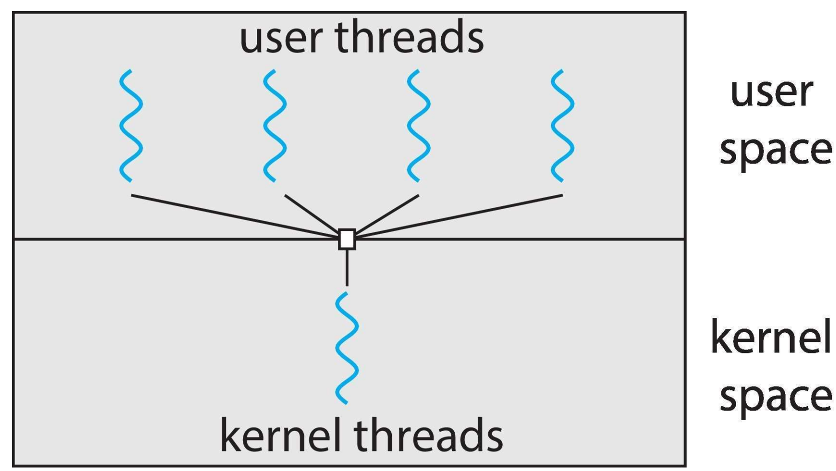


- Many-to-Many == Combined approaches



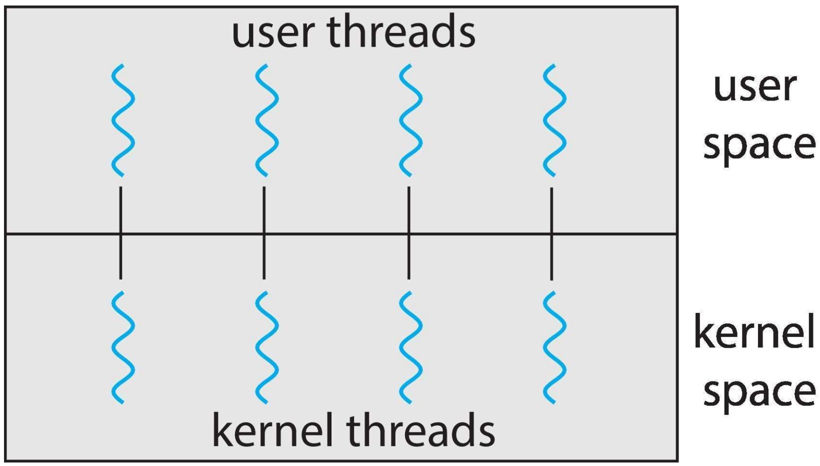
Many-to-One

- 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**



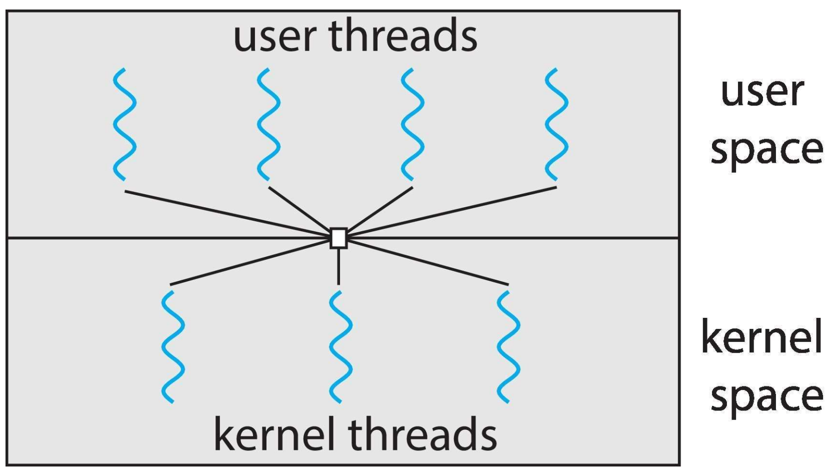
One-to-One

- 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
  - Linux



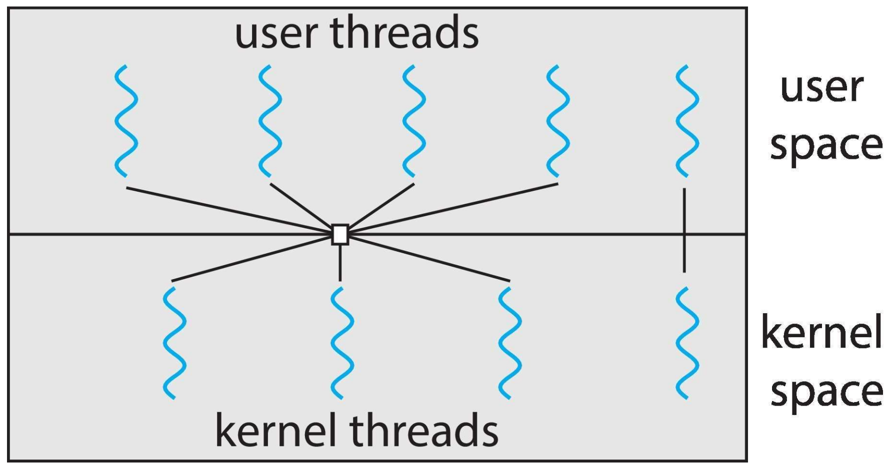
Many-to-Many Model

- 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



Thread Libraries
Multithread Programing

- **Thread library** provides programmer with API for creating and managing threads
- Pthreads library:
  - Common in UNIX-like operating systems (Linux, macOS, Solaris)
- Win32 threads
- Java threads (threads in application-level)
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***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);
}
```

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);
```

End of Chapter 4