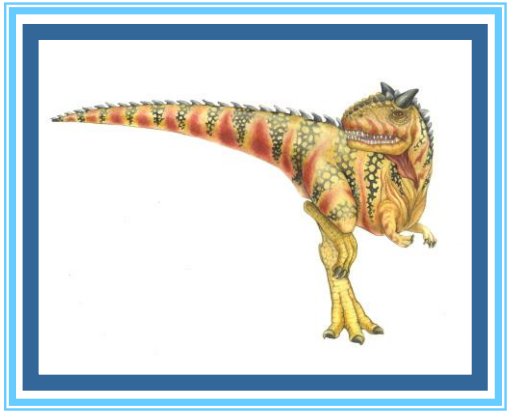
Xi'an Jiaotong-Liverpool University
西交利物浦大學

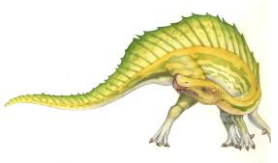
CPT104 - Operating Systems Concepts

Threads



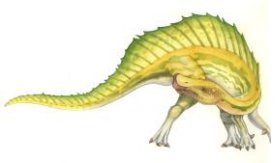
Threads

- Overview / What is a thread?
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit threading
- Threading issues / Designing multithreaded programs



Threads

- **Overview / What is a thread?**
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit threading
- Threading issues / Designing multithreaded programs



Overview

CPU (central processing unit) is a piece of hardware in a computer that executes binary code.

OS (operating system) is software that schedules when programs can use the CPU.

Process is a program that is being executed.

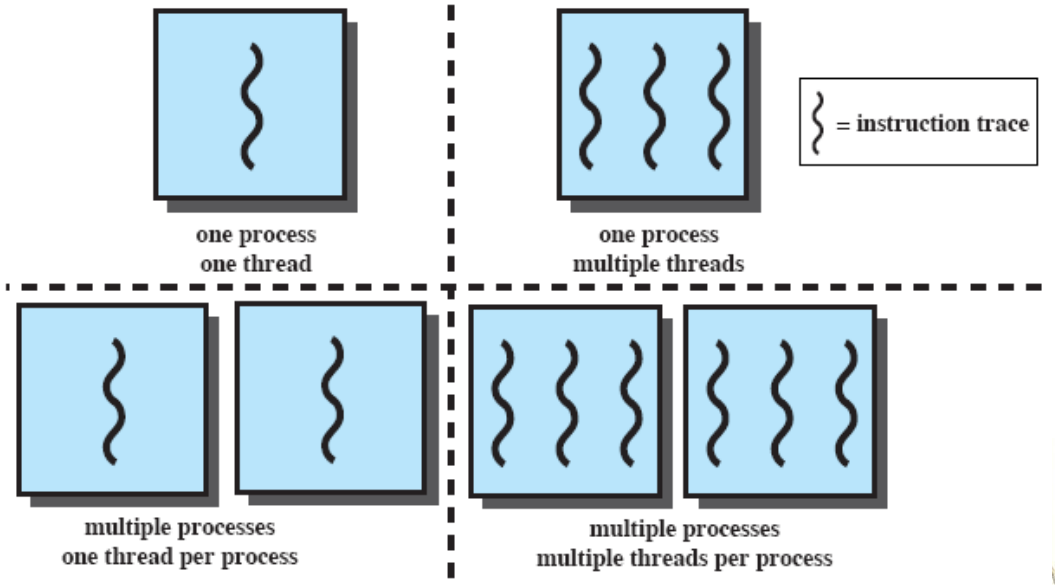
Thread is part of a process.

What is a thread?

- **Thread** of execution is the **smallest sequence of programmed instructions** that can be managed independently by a scheduler and executed by the CPU independently of the parent process.
- A thread is a *lightweight process* that can be managed independently by a scheduler.
- Executes a series of instructions in order (only one thing happens at a time).
- A thread is a *flow of control within a process*.
- When there are multiple threads running for a process, the process provides common memory.
- **Multiple tasks with the application can be implemented by separate threads.**

What is a thread?

- **O.S. view**: a thread is an independent stream of instructions that can be scheduled to run by the OS.
- **Software developer view**: a thread can be considered as a "procedure" that runs independently from the main program.
  - *Sequential program*: a single stream of instructions in a program.
  - *Multi-threaded program*: a program with multiple streams (are needed to use multiple cores/CPUs)



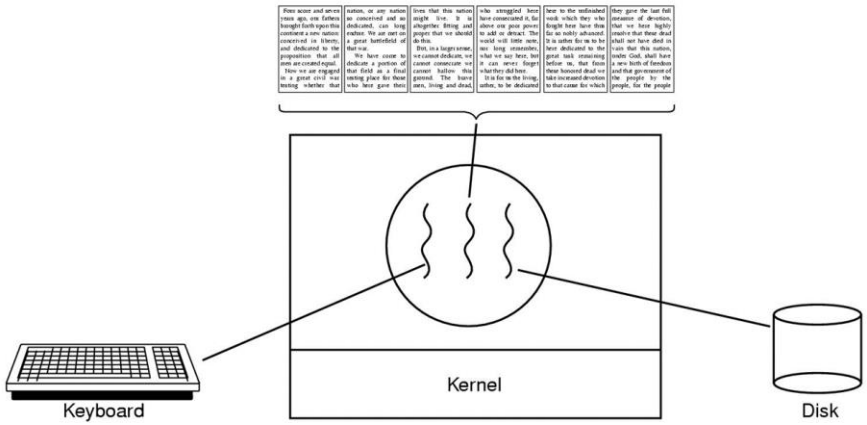
Benefits of Threads

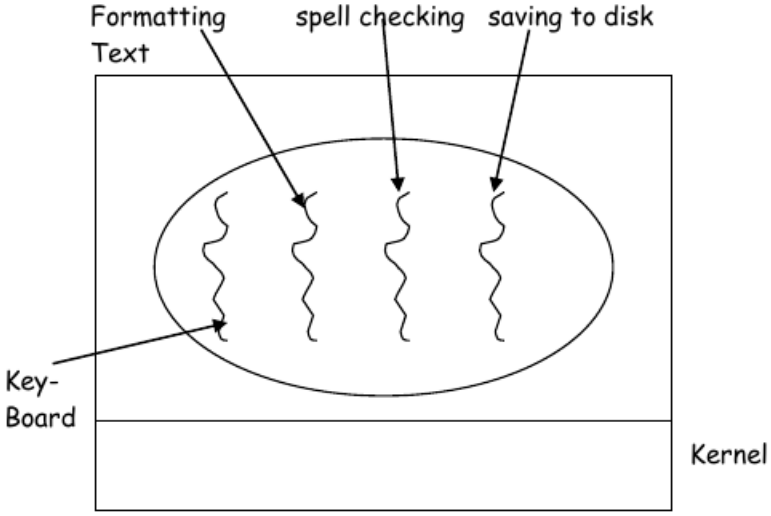
- Responsiveness
  - Allows a program continue running if part of it is blocked or its is performing a lengthy operation
- Resource Sharing and Economy
  - Threads share an address space, files, code, and data
  - Avoid resource consumption
  - Perform much a faster context switch
- Scalability
  - Place code, files and data in the main memory.
  - Distribute threads to each of CPUs, and
  - Threads may be running in parallel on different processing cores.
- Deadlock avoidance

Example of threads

User types text in the Word editor -> **threads**:

- **open a file** in Word editor
- **typing the text** (one thread),
- **the text is automatically formatting** (another thread**),**
- **the text is automatically specifying the spelling mistakes** (another thread)
- **the file is automatically saved to disk** (another thread).

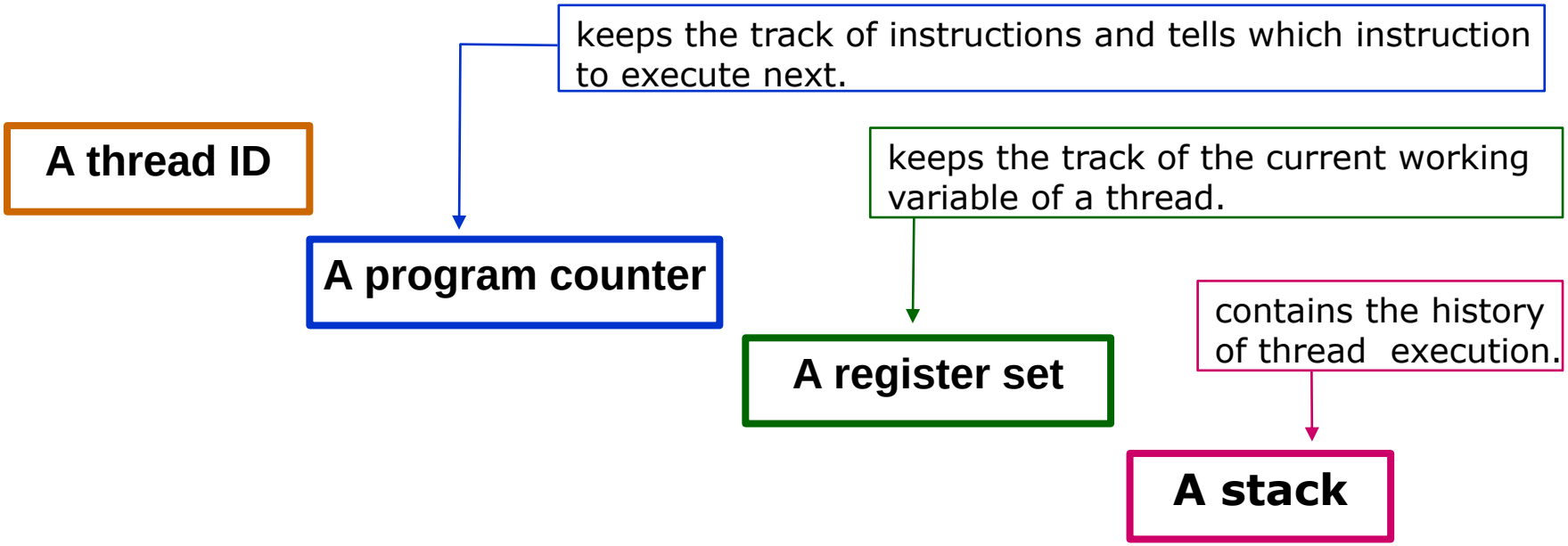




Threads

Threads are scheduled on a processor, and each thread can execute a set of instructions independent of other processes and threads.

Thread Control Block TCB stores the information about a thread



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*.

Thread states

Thread Control Block (TCB) also contains:

Execution State: CPU registers, program counter, pointer to stack

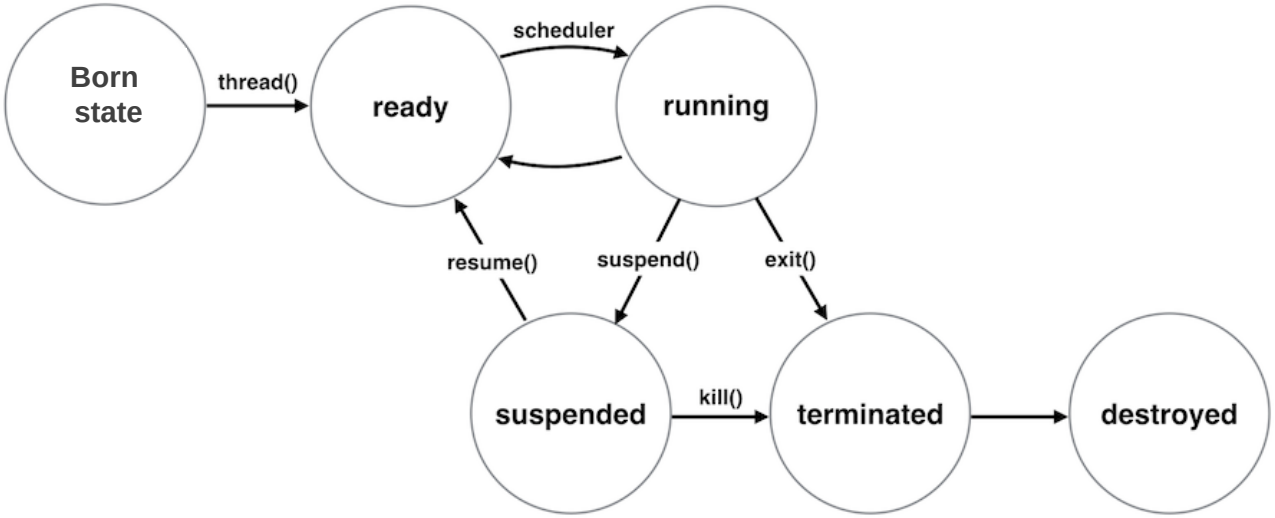
Scheduling info: State, priority, CPU time

Various Pointers (for implementing scheduling queues)

- Pointer to enclosing process (PCB)?

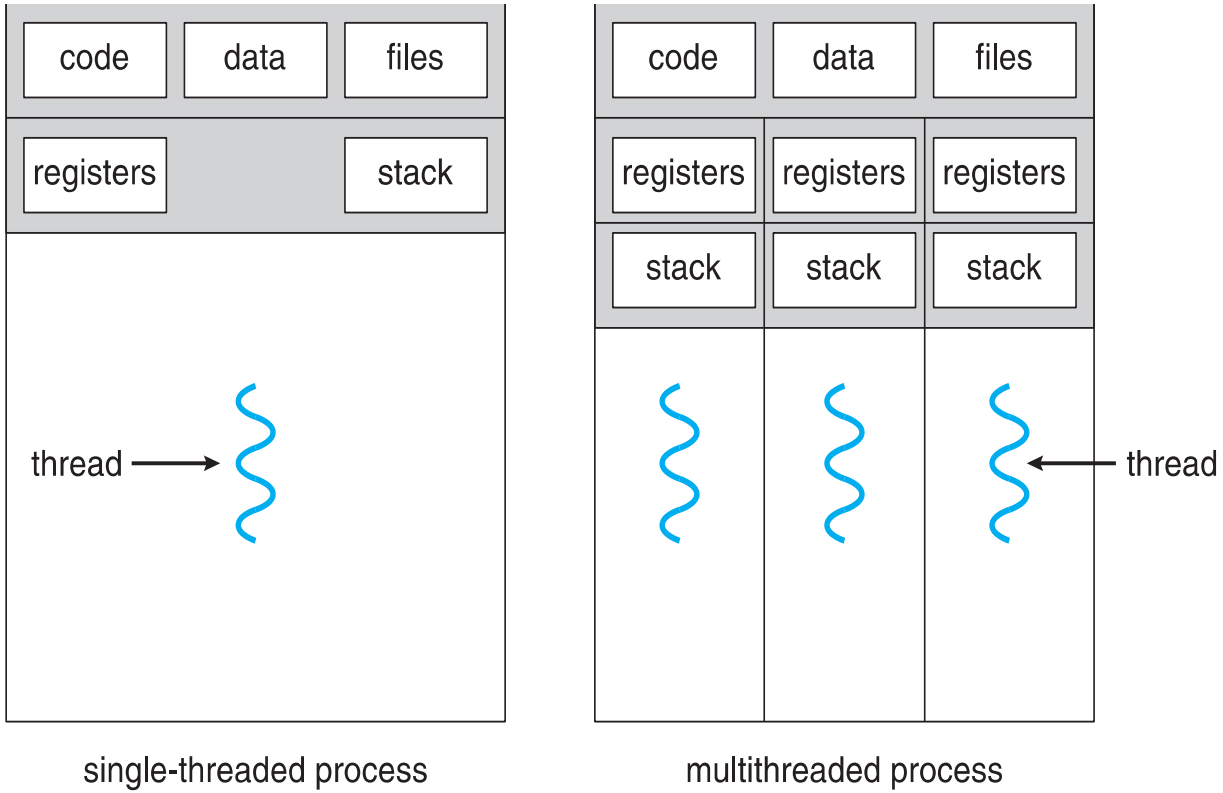
Etc.

OS keeps track of TCBs in protected memory (in Array, or Linked List)



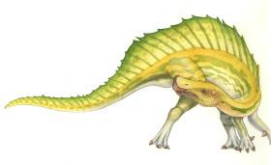
Types of threads per process

- **Single thread process**: Only one thread in an entire process
- **Multi-thread process**: Multiple threads within an entire process (can perform more than one task at time).



Threads

- Overview
- **Multicore Programming**
- Multithreading Models
- Thread Libraries
- Implicit threading
- Threading issues / Designing multithreaded programs

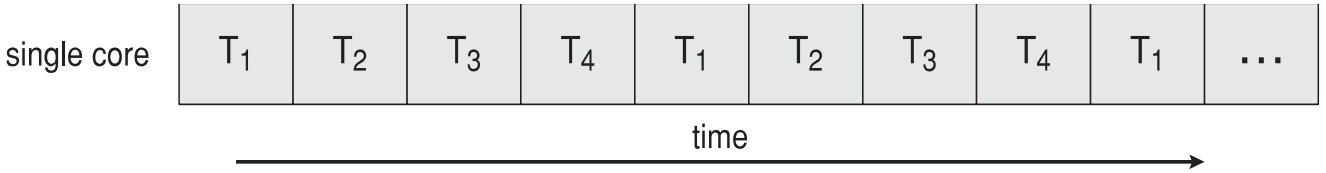


Concurrency vs. Parallelism

Concurrent execution on single-core system

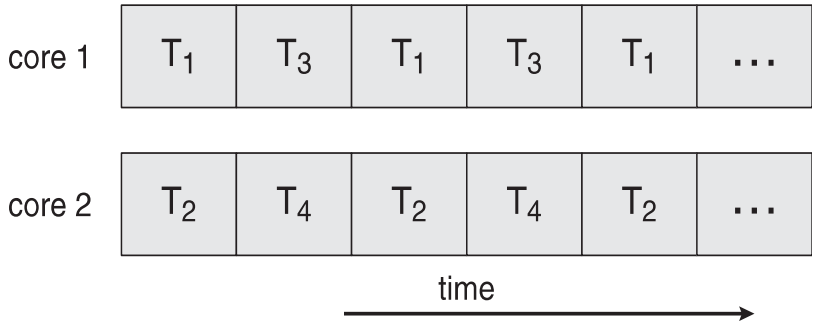
Concurrency means multiple tasks which start, run, and OS rapidly switches back and forth between them -> tasks are **interleaved**

The concurrency creates the illusion of parallel execution.



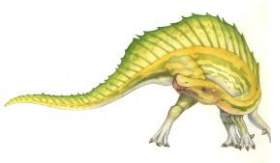
Parallelism on a multi-core system

A system is parallel if it can perform more than one task simultaneously.



Threads

- Overview
- Multicore Programming
- **Multithreading Models**
- Thread Libraries
- Implicit threading
- Threading issues / Designing multithreaded programs



Multithreading Models

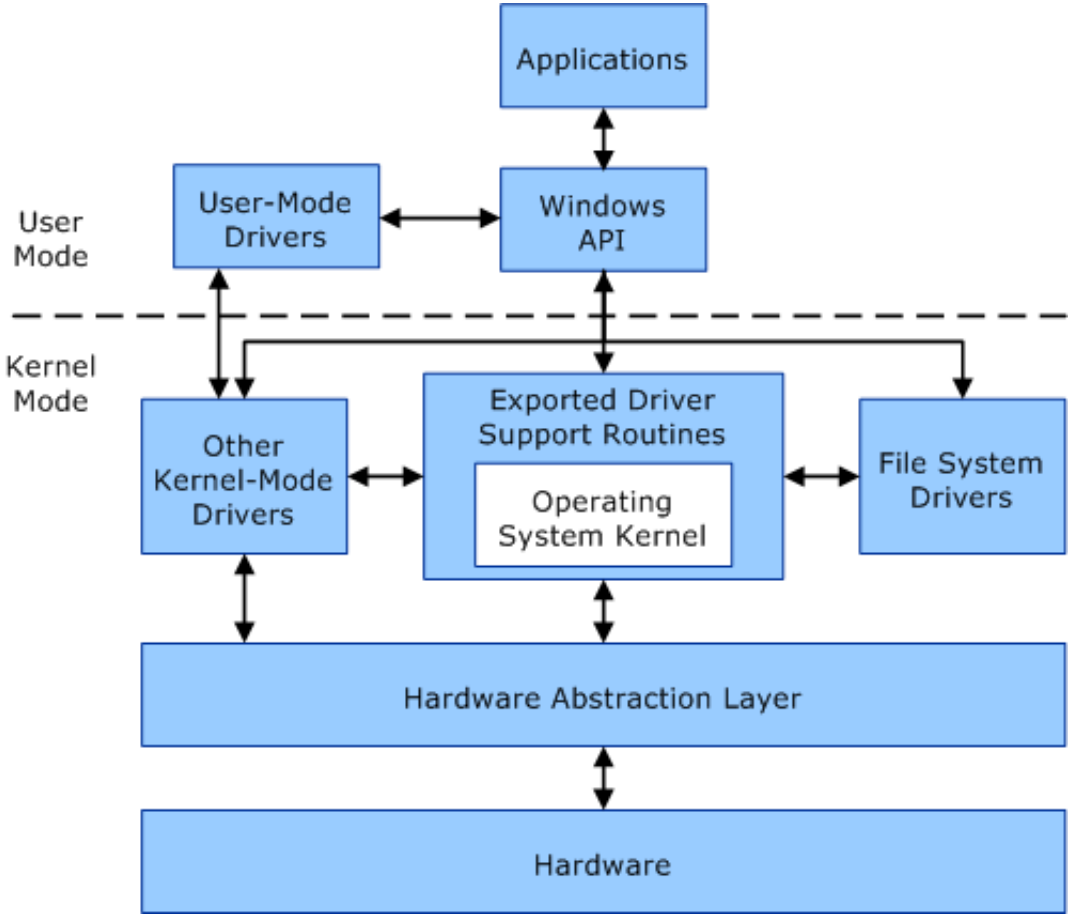
User mode and Kernel mode

The processor switches between the two modes depending on what type of code is running on the processor.

Applications run in user mode.

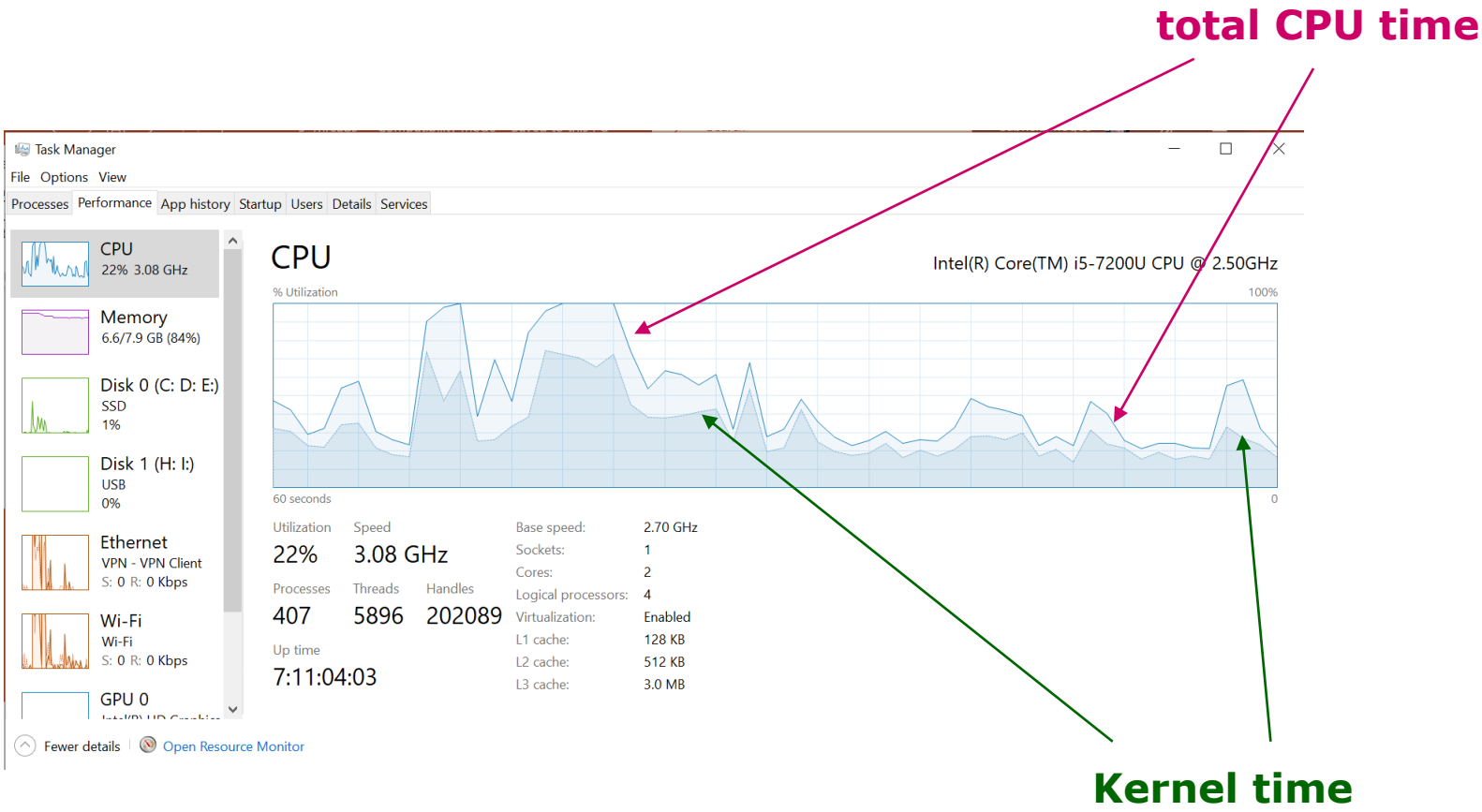
Core operating system components run in kernel mode.

While many drivers run in kernel mode, some drivers may run in user mode.



How can I see my CPU?

In Windows, use **Task Manager** (CTRL+ALT+DEL).

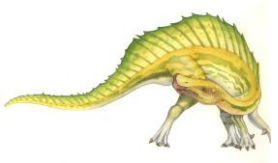


The gap between the two is **User time**.

Types of threads

There are two categories of thread implementation:

- User Level Threads
- Kernel Level Threads



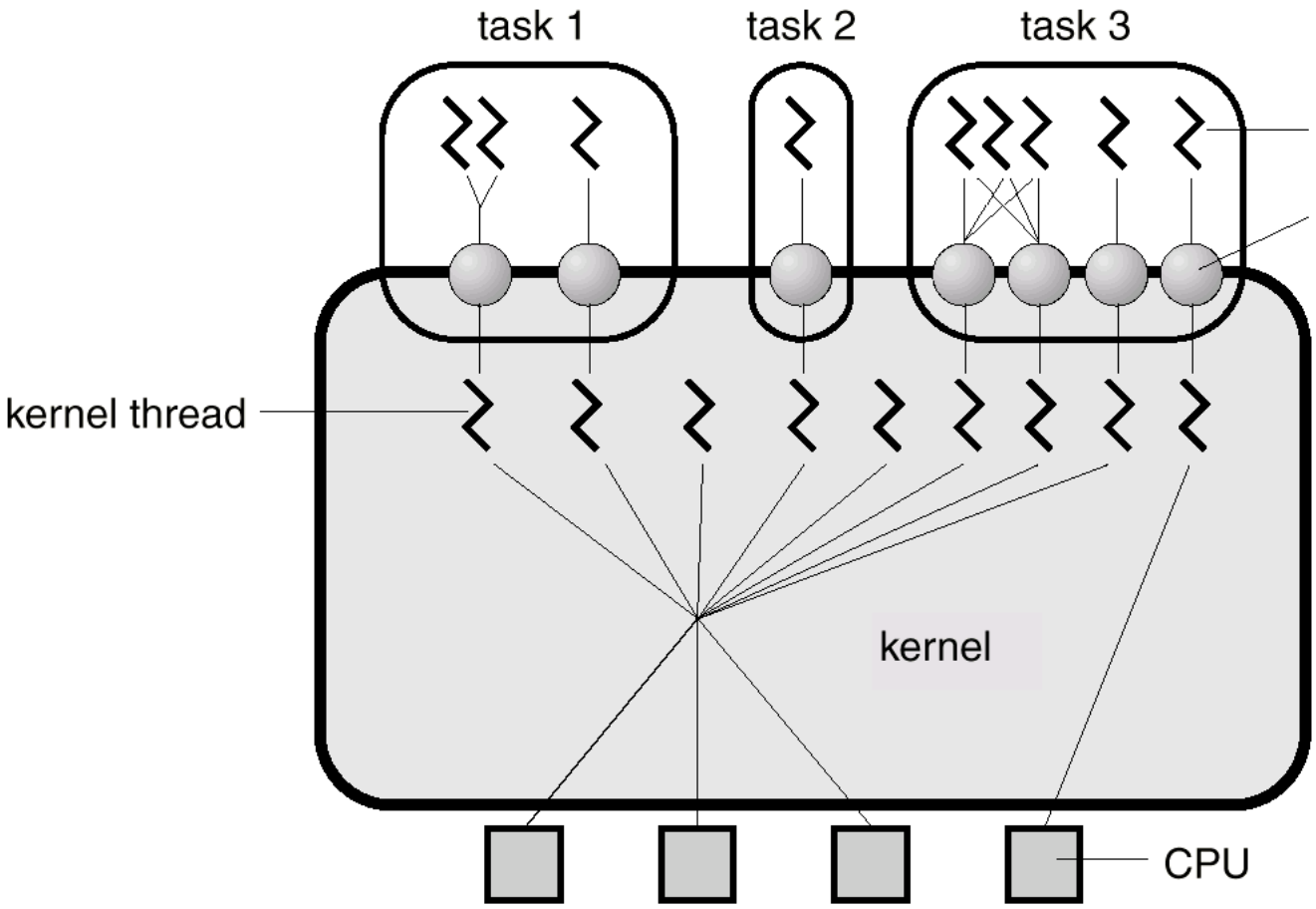
User Threads and Kernel Threads

- **User threads (ULT)** - *is the unit of execution that is implemented by users and the kernel is not aware of the existence of these threads.* Management done by user-level threads library
  - Three primary thread libraries:
    - POSIX **Pthreads**
    - Windows threads
    - Java threads
    - threading in programming like in C#, Python
- **Kernel threads (KLT)** - *are handled by the operating system directly and the thread management is done by the kernel*.
  - *Examples* – virtually all general-purpose operating systems, including:
    - Windows
    - Linux
    - Mac OS X
    - iOS
    - Android

Difference between ULT & KLT

User Level Threads	Kernel Level Thread
User level threads are faster to create and manage.	Kernel level threads are slower to create and manage.
Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads
User level thread is generic and can run on any operating system	. Kernel level thread is specific to the operating system.
Multi-threaded application cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

Example: Solaris Threads



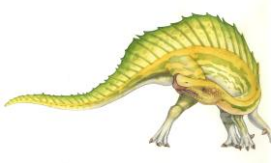
layer between kernel threads and user threads

each LWP supports one or more ULTs and **maps to exactly one KLT**

Task 2 is equivalent to a pure ULT approach
Tasks 1 and 3 map one or more ULT's onto a fixed number of LWP's (&KLT's)
Note how task 3 maps a single ULT to a single LWP bound to a CPU

Multithreading Models

- *Mapping* user level threads to kernel level threads
- In a combined system, multiple threads within the same application can run in parallel on multiple processors.
- **Multithreading models** are three types
  - **Many – to – One**
  - **One – to – One**
  - **Many – to - Many**



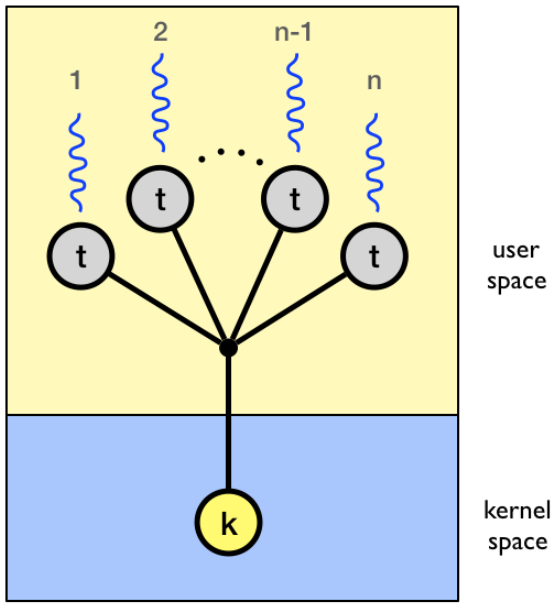
Many-to-One Model

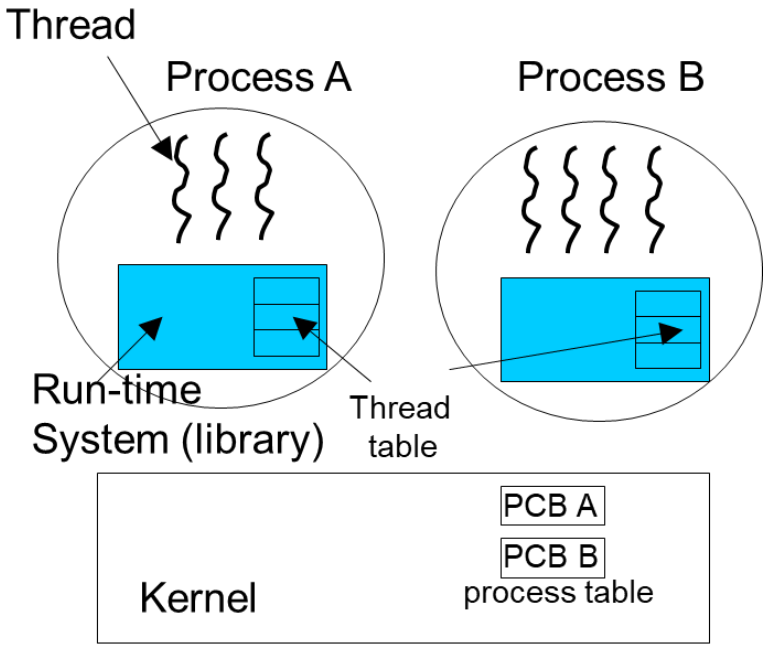
Many user-level threads are mapped to a single kernel thread

- The process can only run one user-level thread at a time because there is only one kernel-level thread associated with the process.
- Thread management done at user space, by a **thread library**

Examples:

- Solaris Green Threads
- GNU Portable Threads





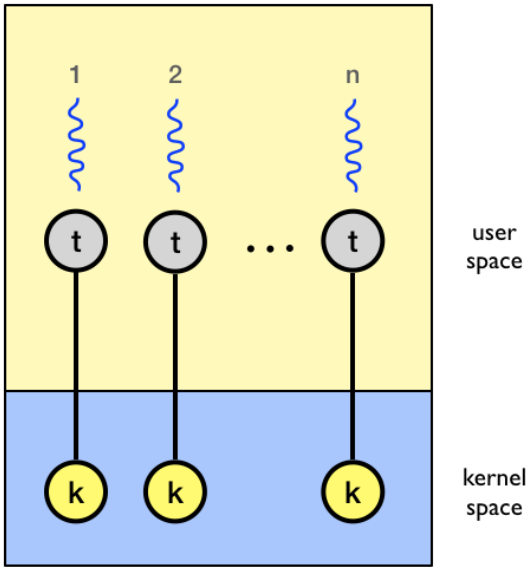
One-to-One Model

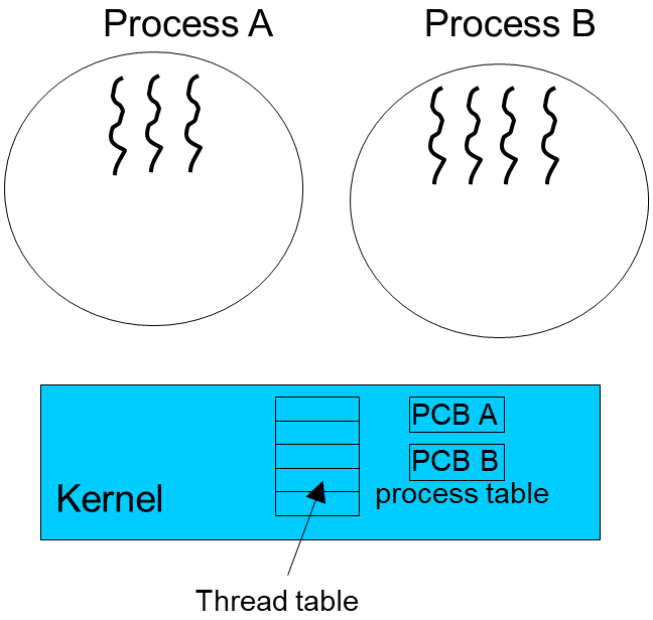
Each user thread mapped to one kernel thread

- Kernel may implement threading and can manage threads, schedule threads.
- Kernel is aware of threads.
- Provides more concurrency; when a thread blocks, another can run.

Examples

- Windows NT/XP/2000
- Linux
- Solaris 9 and later





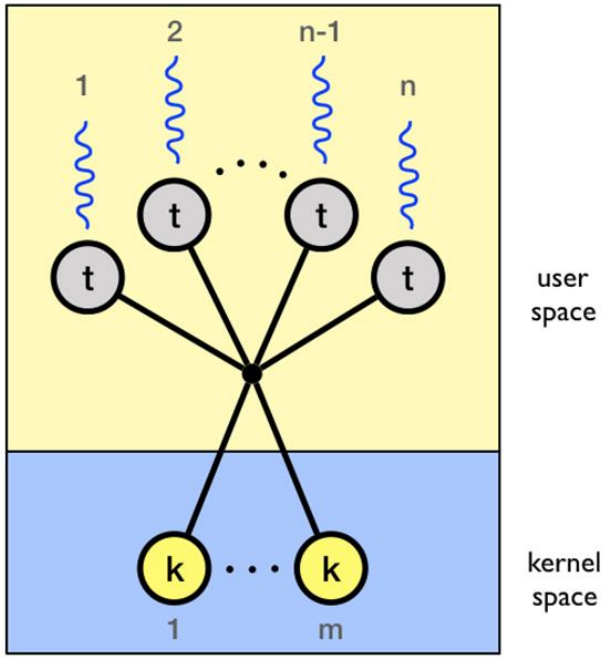
Many-to-Many Model

Allows many user level threads to be mapped to many kernel threads

- Allows the operating system to create **a sufficient number of kernel threads**
- Number of kernel threads may be specific to an either a particular application or a particular machine.
- The user can create any number of threads and corresponding kernel level threads can run in parallel on multiprocessor.

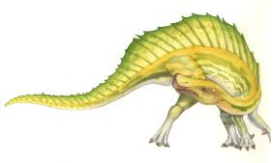
Example:

- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package
- Solaris older than Solaris 9 - *Two-level Relationship Multithreading Model*



Threads

- Overview
- Multicore Programming
- Multithreading Models
- **Thread Libraries**
- Implicit threading
- Threading issues / Designing multithreaded programs



Thread Libraries

Threads can be created, used, and terminated via a set of functions that are part of a **Thread API** (a **thread library**).

Thread library provides programmer with **API** (*Application Programming Interface*) for creating and managing threads

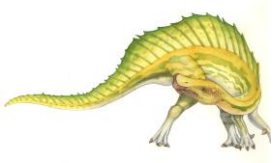
- **Threads may be implemented in *user space* or *kernel space*.**

- **Library** entirely **in user space** *with no kernel support.*
  - *all code and data structures for the library exist in user space*.
  - invoking a function in the library results in a *local function call in user space* and not a system call.
- **Kernel-level library** *supported directly by the operating system*.
  - *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*.

Three primary thread libraries: **POSIX threads**, **Java threads**, **Win32 threads**

Threads

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- **Implicit threading**
- Threading issues / Designing multithreaded programs

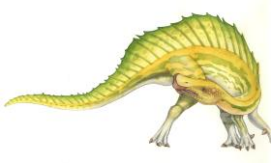


Managing threads

There are 2 categories: **Explicit** and **Implicit threading**.

Explicit threading - the *programmer* creates and manages threads**.**

Implicit threading - the *compilers and run-time libraries* create and manage threads**.** *We are interested on it!*



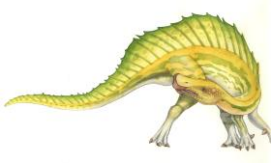
Implicit threading

Three alternative approaches for **designing multithreaded programs**:

- **Thread pool** - create a number of threads at process startup and place them into a pool, where they sit and wait for work.
- **OpenMP** is a set of compiler directives available for C, C++, and Fortran programs that instruct the compiler to automatically generate parallel code where appropriate.
- **Grand Central Dispatch (GCD)** - is an extension to C and C++ available on *Apple's MacOS X* and *iOS* operating systems to support parallelism.

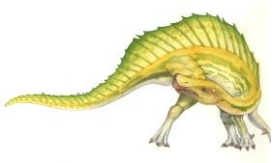
Threads

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit threading
- **Threading issues / Designing multithreaded programs**



Threading Issues in Multithreading environments

1. **fork()** and **exec() System Calls**
2. **Signal Handling**
3. **Thread Cancellation**



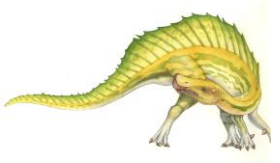
1. fork() and exec() System Calls

Does **fork()** duplicate only the calling thread or all threads?

- **fork()** would create a new duplicate process.

Here the issue is whether the new duplicate process created by fork() will duplicate all the threads of the parent process or the duplicate process would be single-threaded.

- **exec**() system call when invoked replaces the program along with all its threads with the program that is specified in the parameter to exec()



2. Signal Handling

Signals are software interrupts which sent to a program to indicate that an important event has occurred.

Signal is used in ***UNIX*** systems.

Windows does not explicitly provide support for signals.

All signals follow the pattern:

1. A signal is generated by the occurrence of a particular event.
2. The signal is delivered to a process.
3. Once delivered, the signal must be handled.

Example 1:

- **illegal memory access** and **division by 0.**

If a running program performs either of these actions, a signal is generated. Signals are delivered to the same process that performed the operation that caused the signal.

Sending Signals To Processes

***Example 2*: Sending Signals Using The Keyboard**

Ctrl-C

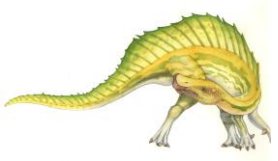
Pressing this key causes the system to send an INT signal (SIGINT) to the running process. By default, this signal causes the process to immediately terminate (interrupt).

Ctrl-Z

Pressing this key causes the system to send a TSTP signal (SIGTSTP) to the running process. By default, this signal causes the process to suspend execution.

Ctrl-

Pressing this key causes the system to send a QUIT signal (SIGQUIT) to the running process. By default, this signal causes the process to immediately terminate (kill).



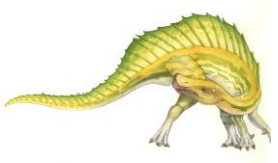
2. Signal Handling

Processes catch and handle signals by running special *signal handler* routines. After the signal handler exits the operating system will restart the process at the point where the process was originally interrupted

A signal may be **handled** by one of two possible handlers:

1. **A default signal handler** - The default action is what will happen if the signal is not caught and handled by the application. The default action for many signal types is to terminate the process immediately. In a few cases the default action is to simply ignore the signal.

2. **A user-defined signal handler** - the action is user-defined.



3. Thread Cancellation

Two general approaches:

- **Asynchronous cancellation** terminates the target thread *immediately*.
  - it is troublesome if a thread to be canceled is in the middle of updating shared data
- **Deferred cancellation** allows the *target thread* to periodically check if it should be cancelled

The issue related to the target threads are listed below:

- What if the resources had been allotted to the cancel target thread?
- What if the target thread is terminated when it was updating the data, it was sharing with some other thread.

End of Lecture

Summary

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit threading
- Threading issues / Designing multithreaded programs

Reading

- **Chapter 4 of the module textbook**