

# Operating Systems Concepts

Threads



CS 4375, Fall 2025

Instructor: MD Armanuzzaman (*Arman*)

[marmanuzzaman@utep.edu](mailto:marmanuzzaman@utep.edu)

September 10, 2025

# Summery

- Introduction to xv6
  - What is xv6?
  - RISC-V
  - Hardware-software stack of xv6
  - xv6 system calls
  - Code-base overview
- Quiz 1

# Quiz 1 Statistics

| Grade Statistics   |      | Grading Status |   | Grade Distribution (%) |    |
|--------------------|------|----------------|---|------------------------|----|
| Grade count        | 60   | Not started    | 2 | Greater than 100       | 0  |
| Minimum value      | 0    | In progress    | 0 | 90 - 100               | 12 |
| Maximum value      | 10   | Needs grading  | 0 | 80 - 89                | 9  |
| Range              | 10   | Exempt         | 0 | 70 - 79                | 18 |
| Average            | 6.75 |                |   | 60 - 69                | 10 |
| Median             | 7    |                |   | 50 - 59                | 4  |
| Standard deviation | 2.19 |                |   | 40 - 49                | 3  |
| Variance           | 4.82 |                |   | 30 - 39                | 1  |
|                    |      |                |   | 20 - 29                | 0  |
|                    |      |                |   | 10 - 19                | 0  |
|                    |      |                |   | 0 - 9                  | 3  |
|                    |      |                |   | Less than 0            | 0  |

# Agenda

- Threads
  - Concurrent programming
  - Why threads?
  - Threads vs Processes
  - Thread pools
  - Threading implementation & multithreading models
  - Threading issues
    - Semantics of `fork()` and `exec()`
    - Thread cancellation
    - Signal handling

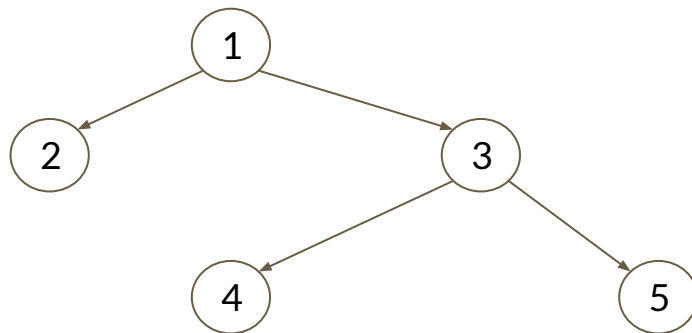
# Concurrent Programming

In certain cases, a single application may need to run several tasks at the same time.

Sequential



Concurrent



# Motivation

- Increase the performance by running more than one task at a time
  - Divide the program to  $n$  smaller pieces, and run it  $n$  times faster using  $n$  processors
- To cope with independent physical devices
  - Do not wait for a blocked device, perform other operations in the background

# Serial vs Parallel



# Divide and Compute

$$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8$$

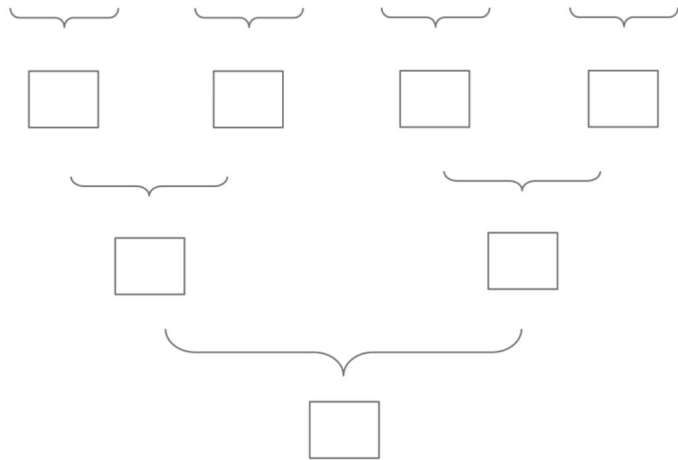
How many operations with sequential programming? 7

- Step 1:  $x_1 + x_2$
- Step 2:  $x_1 + x_2 + x_3$
- Step 3:  $x_1 + x_2 + x_3 + x_4$
- Step 4:  $x_1 + x_2 + x_3 + x_4 + x_5$
- Step 5:  $x_1 + x_2 + x_3 + x_4 + x_5 + x_6$
- Step 6:  $x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7$
- Step 7:  $x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8$



# Divide and Compute

$$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8$$



- **Step 1:** Parallelism = 4
- **Step 2:** Parallelism = 2
- **Step 3:** Parallelism = 1

# Gain From Parallelism

- In theory:
  - Dividing a program into  $n$  smaller parts and running on  $n$  processors results in  $n$  time speedup
- In practice:
  - This is not true, due to
    - Communication costs
    - Dependencies between different program parts
      - The previous addition example can run only in  $\log(n)$  time, and not  $1/n$

# Concurrent Programming

- Implementation of concurrent tasks:
  - As separate programs
  - As a set of processes or threads created by a single program
- Execution of concurrent tasks:
  - On a single processor (can be multiple cores)
    - Multithreaded programming
  - On several processors in close proximity
    - Parallel computing
  - On several processors distributed across a network
    - Distributed computing

# Why Threads?

In certain cases, a single application may need to run several tasks at the same time

- Creating a new process for each task is time and resource consuming
- Use a single process with multiple threads
  - Faster
  - Less overhead for creation, switching, and termination
  - Share the same address space

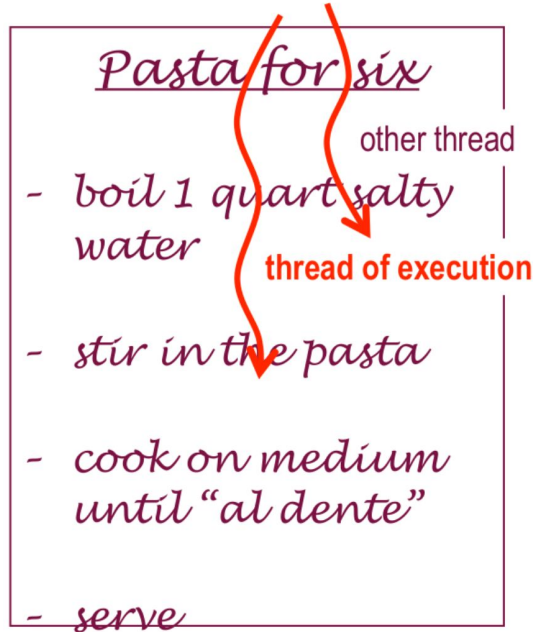
# Threading Benefits

Patterns of multithreading usage across applications:

- Perform foreground and background work in parallel
  - Illusion of full-time interactivity toward the user while performing other tasks (same principle as **time-sharing**)
- Allow asynchronous processing
  - Separate and desynchronize the execution streams of independent tasks that don't need to communicate.
  - Handle external, surprise events such as client requests
- Increase speed of execution
  - “Stagger” and overlap CPU execution time and I/O wait time (same principle as **multiprogramming**)

# Multithreading

The execution part is a “thread” that can be multiplied

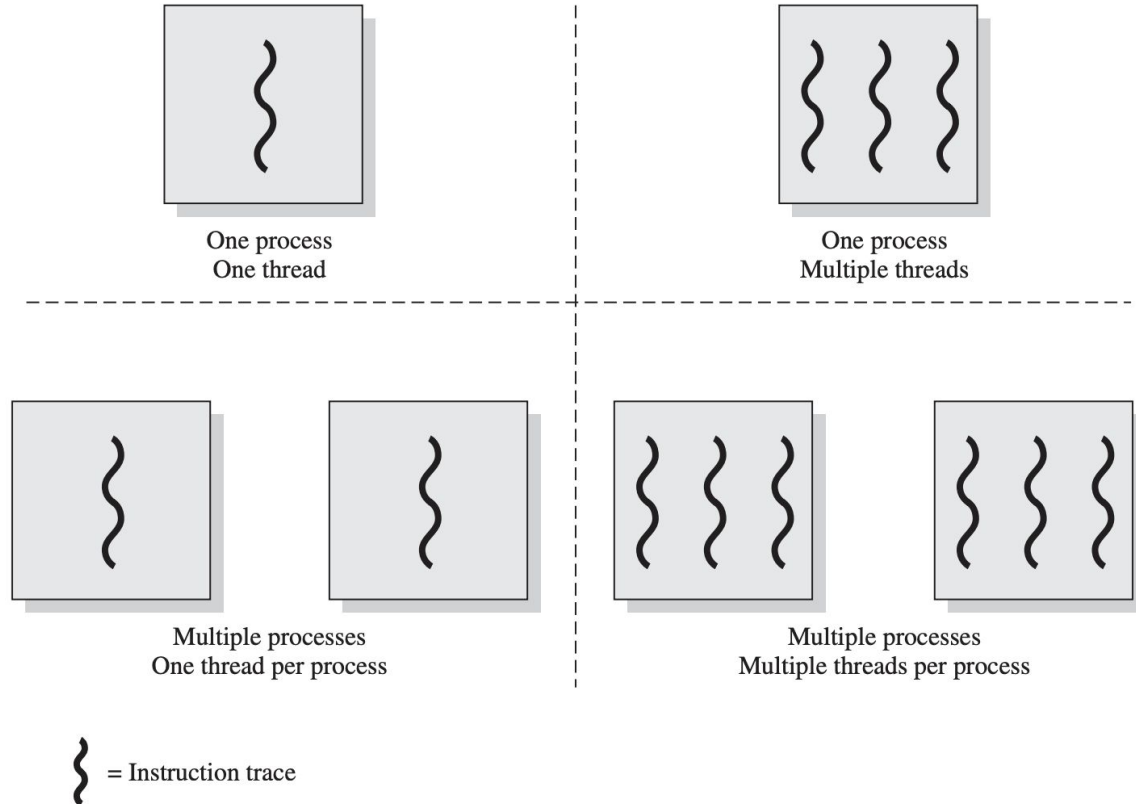


Program

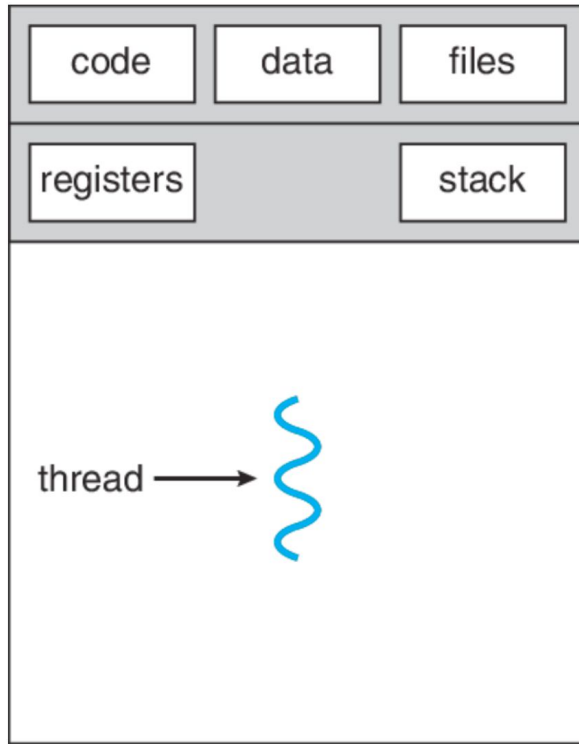


Process

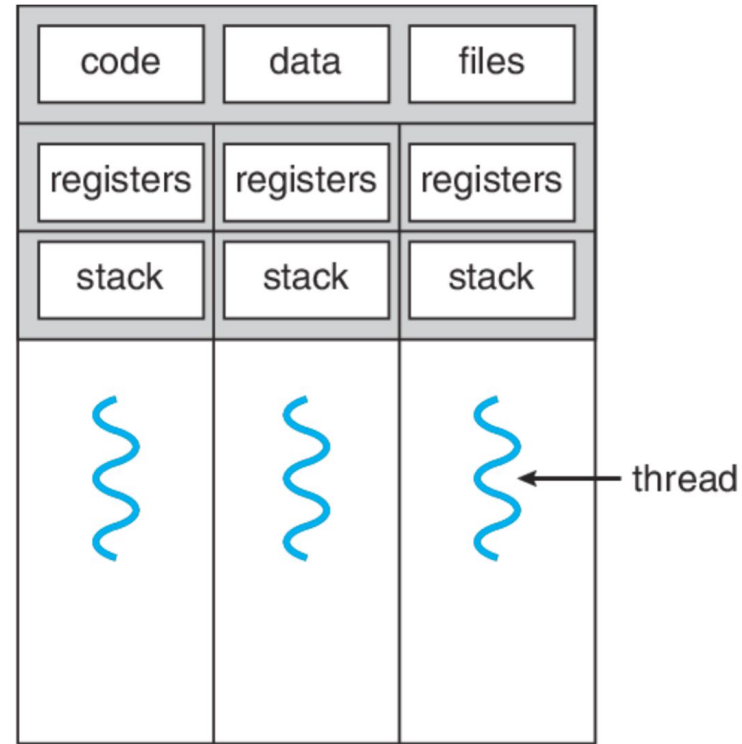
# Multithreading



# Single and Multithreaded Processes



single-threaded process



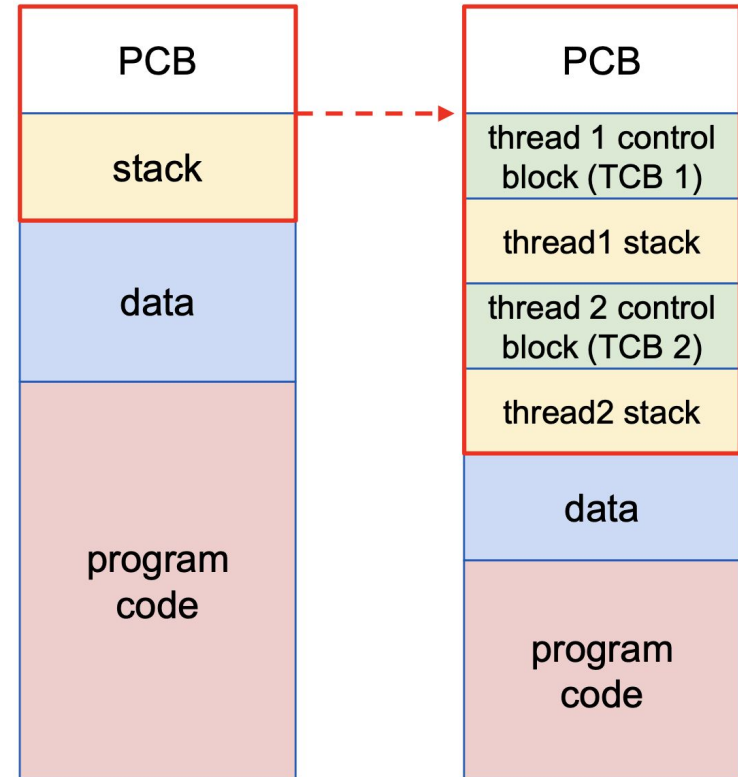
multithreaded process



# New Process Description Model

Multithreading requires changes in the process description model.

- Each thread of execution receives its own control block and stack.
  - Own execution state (“Running”, “Blocked”, etc.)
  - Own copy of CPU registers
  - Own execution history (stack)
- The process keeps a global control block listing resources currently used



# Per-process vs Per-thread Items

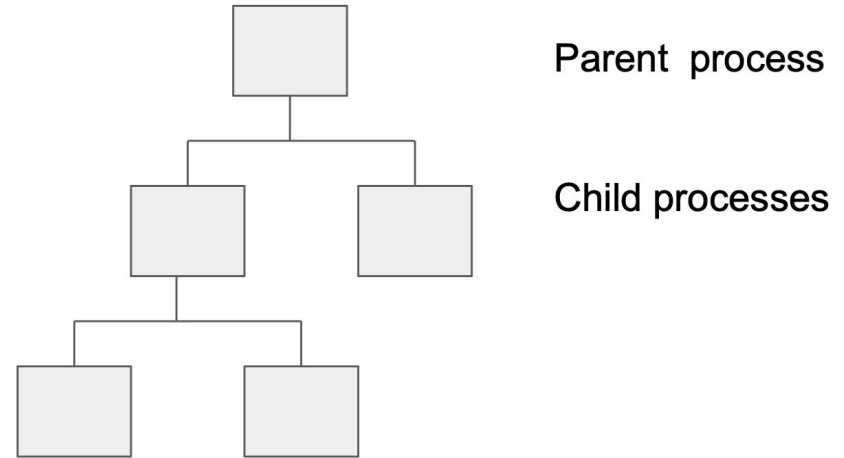
Per-process items and **per-thread items** in the control block structures:

- process identification data + **thread identifiers**
  - Numeric identifiers of the process, the parent process, the user, etc.
- **CPU state information**
  - User-visible, control & status registers
  - Stack pointers
- Process control information
  - **Scheduling: state, priority, awaited event**
  - Used memory and I/O, opened files, etc.
  - Pointer to next PCB

# Multiprocessing Model

## Process spawning

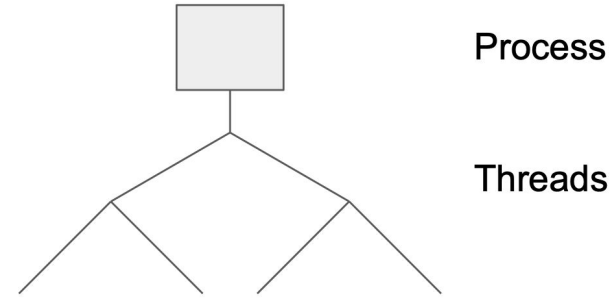
- Setting up the process control block
- Allocation of an address space
- Loading the program into the allocated address space
- Passing on the process control block to the scheduler



# Multiprocessing Model

## Thread spawning

- Threads are created *within* processes, and belonging to processes
- All the threads created within one process, share the resources of the process including the address space
- Scheduling is performed on per-thread basis
- Threads have a similar lifecycle as the processes and will be managed mainly in the same way



# Threads vs Processes

- A common terminology:
  - Heavyweight Process = Process
  - **Lightweight Process = Thread**
- **Advantages:**
  - Much quicker to create a thread than a process
    - spawning a new thread only involves allocating a new stack and a new CPU state block
  - Much quicker to switch between threads than to switch between processes
  - Threads share data easily
- **Disadvantages:**
  - Processes are more flexible
    - They don't have to run on the same processor
  - No security between threads: One thread can stomp on another thread's data
  - For threads which are supported by user thread package instead of the kernel:
    - If one thread blocks all threads in task block

# Example

Consider a process with two concurrent threads T1 and T2. The code being executed by T1 and T2 is as follows:

Shared Data:

X = 5;

Y = 10;

T1:

Y = X + 1;

X = Y;

Print X;

T2:

U = Y - 1;

Y = U;

Print Y;

Assume that each assignment statement on its own is executed as an *atomic* operation.

What are the possible outputs of this process?

# Solution

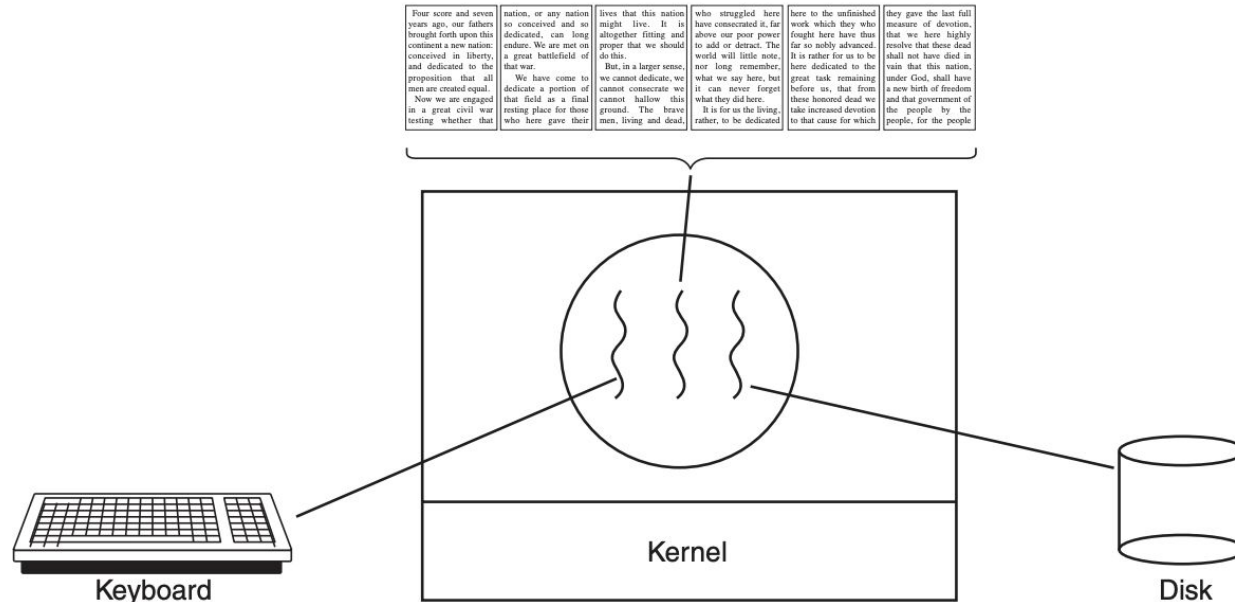
All six statements can be executed in any order!

Possible outputs are:

1. 65
2. 56
3. 55
4. 99
5. 66
6. 69
7. 96

# Threading Examples

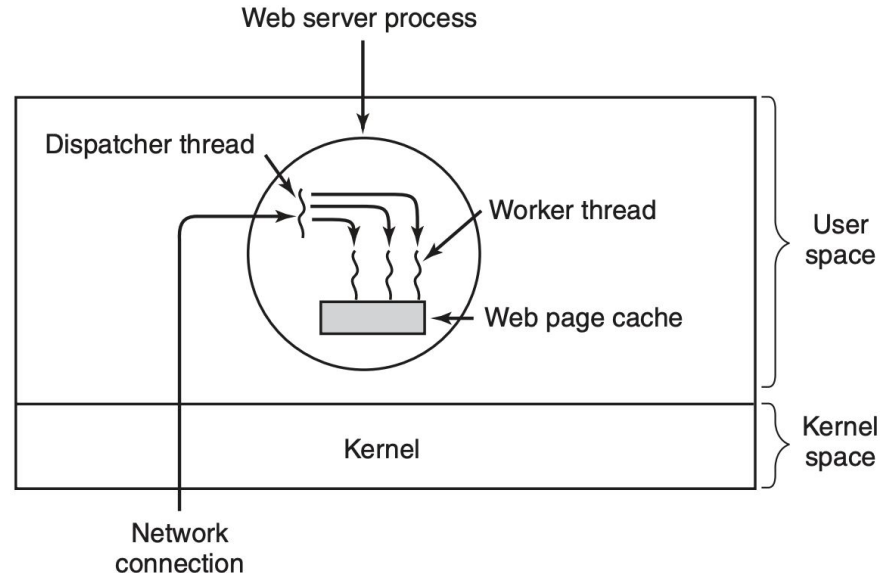
- Word processor:
  - One thread listens continuously to keyboard and mouse events to refresh the GUI; a second thread reformats the document (to prepare page 600); a third thread writes to disk periodically.





# Threading Examples

- Web server:
  - As each new request comes in, a “dispatcher thread” spawns a new “worker thread” to read the requested file (worker threads may be discarded or recycled in a “thread pool”)



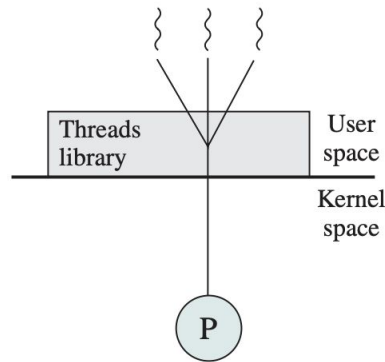
# Thread Pools

- Threads come with some overhead as well
- Unlimited threads can exhaust system resources, such as CPU or memory
- Create a number of threads at process startup and put them in a pool, where they await work
- When a server receives a request, it awakens a thread from this pool
- Advantages:
  - Usually faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application to be bound to the size of the pool
- Number of threads in the pool can be setup according to:
  - Number of CPUs, memory, expected number of concurrent requests

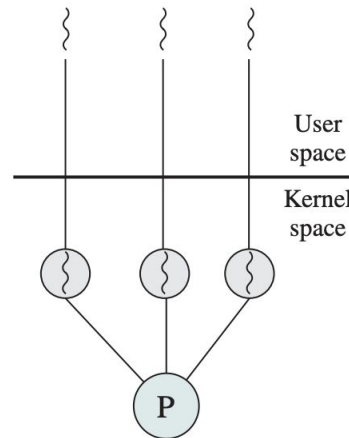
# Thread Implementation

Two broad categories of thread implementation:

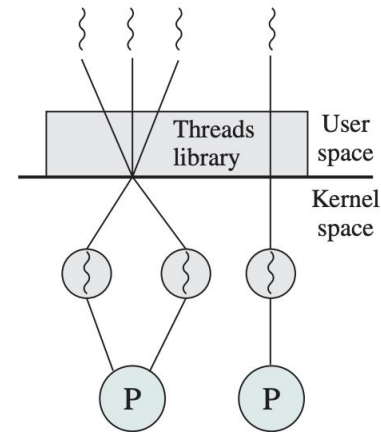
- User-Level Threads (ULTs)
- Kernel-Level Threads (KLTs)



(a) Pure user-level



(b) Pure kernel-level



(c) Combined

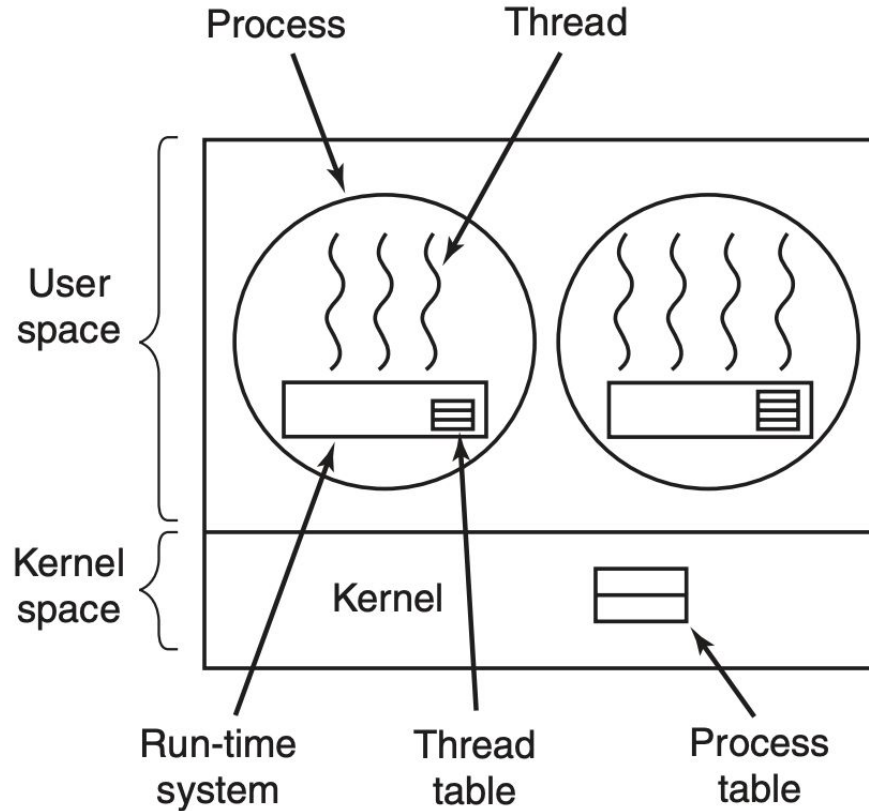
 User-level thread    Kernel-level thread    Process

# Thread Implementation - ULT

## User-Level Threads (ULTs):

- The kernel is not aware of the existence of threads, it knows only processes with one thread of execution (one PC)
- Each user process manages its own private thread table
- Pros:
  - **Light thread switching:** Does not need kernel mode privileges
  - **Cross-platform:** ULTs can run on any underlying OS
- Cons:
  - **If a thread blocks, the entire process is blocked**, including all other threads in it

# Thread Implementation - ULT

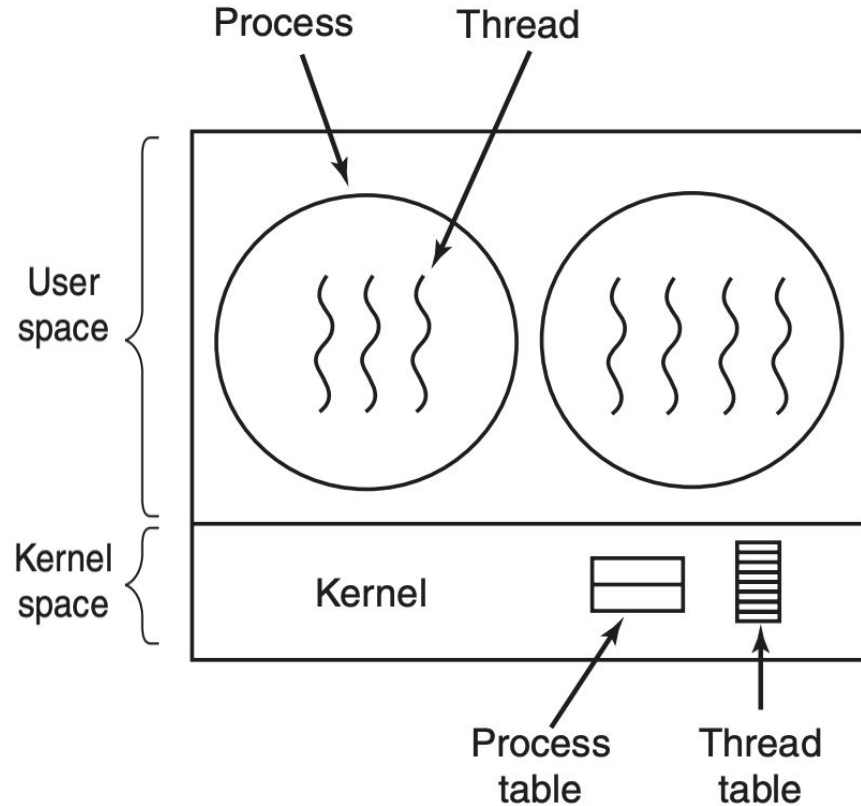


# Thread Implementation - KLT

## Kernel-Level Threads (KLTs):

- The kernel knows about and manages all system calls
- Supported by most operating systems (Linux, Windows, Mac OS X, Solaris, ...)
- Pros:
  - Fine-grain scheduling, done on a thread basis
  - If a thread blocks, another one can be scheduled without blocking the whole process
- Cons:
  - Heavy thread switching involving mode switch

# Thread Implementation - KLT



# Different Multithreading Models

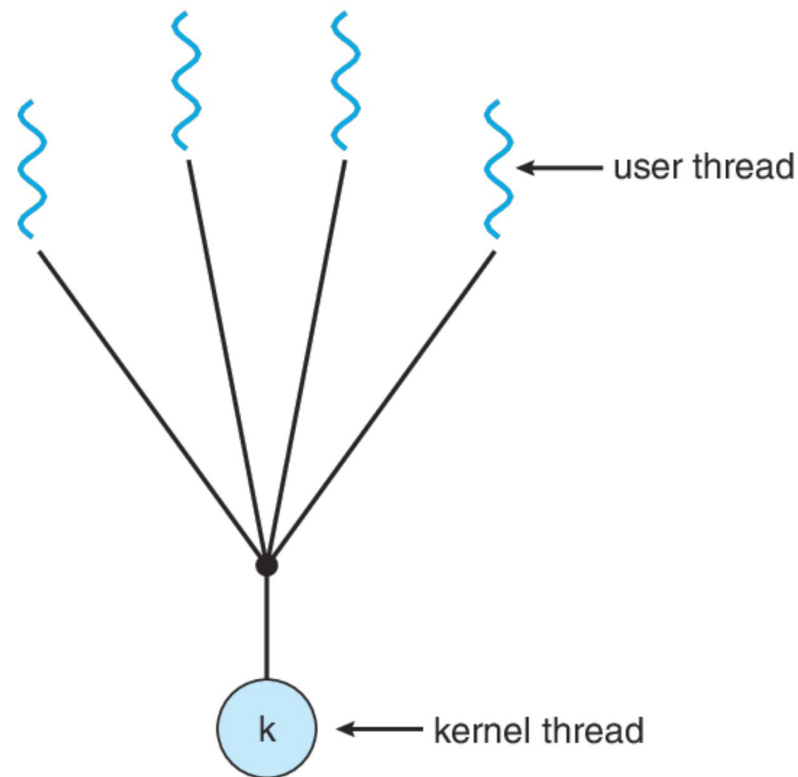
- Many-to-One
- One-to-One
- Many-to-Many
- Hybrid (Two-level)



# Different Multithreading Models

- Many-to-One

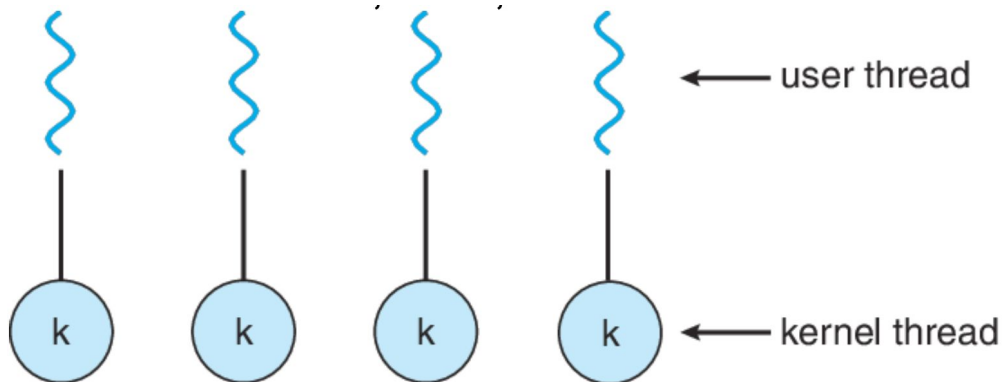
- Several user-level threads mapped to single kernel thread
- Thread management in user space
  - efficient
- If a thread blocks, **entire process blocks**
- One thread can access the kernel at a time
  - limits parallelism
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



# Different Multithreading Models

- One-to-One

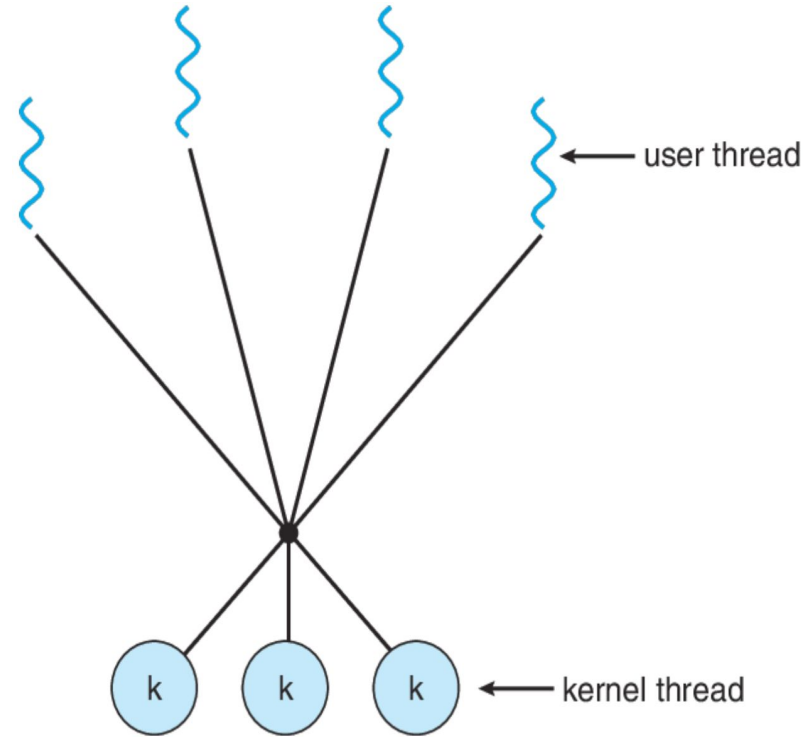
- Each user-level thread maps to a kernel thread
- A blocking thread does not block other threads
- Multiple threads can access kernel concurrently → increased parallelism
- **Drawback:** Creating a user level thread requires creating a kernel level thread → increased overhead and limited number of threads
- Examples: Windows NT/XP/2000, Linux, Solaris 9 and later



# Different Multithreading Models

- Many-to-Many

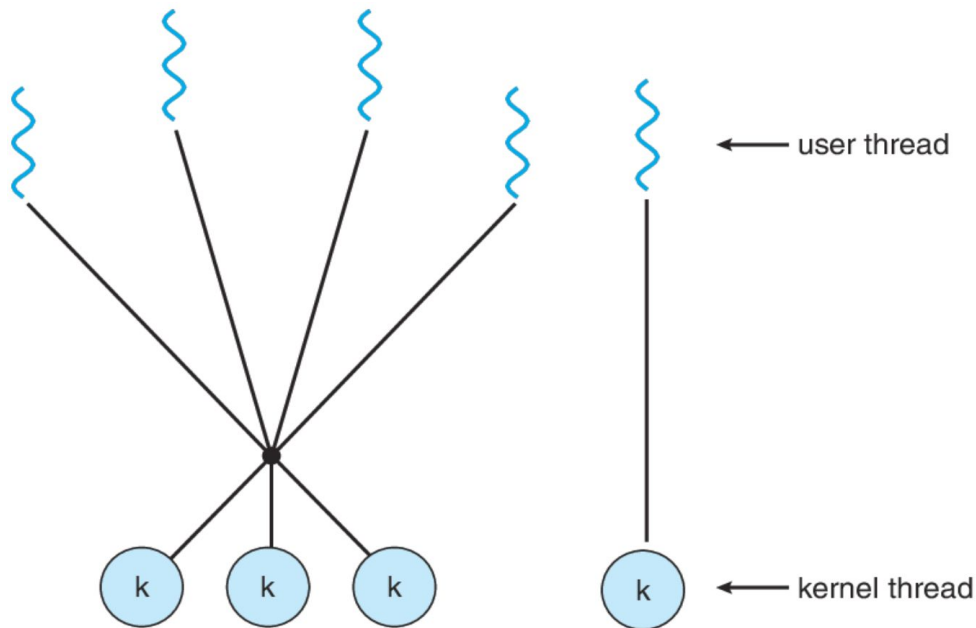
- Allows many user level threads to be mapped to a smaller number of kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Increased parallelism as well as efficiency
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package



# Different Multithreading Models

- Hybrid (Two-level)

- Similar to Many-to-Many, except that it allows a user thread to be bound to kernel thread.
- Examples: IRIX, HP-UX, Tru64 UNIX, Solaris 8 and earlier



# Threading Issues

- Semantics of `fork()` and `execvp()` system calls
- Thread cancellation
- Signal handling

# Semantics of fork() and exec()

Semantics of *fork()* and *exec()* system calls change in a multithreaded program

- E.g. if one thread in a multithreaded program calls fork()
  - Should the new process duplicate all threads?
  - Or should it be single-threaded?
- Some UNIX systems implement two versions of fork()
- If a thread executes exec() system call
  - Entire process will be replaced, including **all threads**

# Thread Cancellation

- Terminating a thread before it has finished
  - If one thread finishes searching a database, others may be terminated
  - If user presses a button on a web browser, web page can be stopped from loading further
- Two approaches to cancel the target thread
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
    - More controlled and safe

# Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- All signals follow this pattern:
  - Signal is generated by particular event
  - Signal is delivered to a process
  - Once delivered, a signal must be handled
- In **multithreaded systems**, there are 4 options:
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process



# Homework

- Submission
  - Code hosted on github (separate branches for each homework)
    - Your solution branch should be private
    - Give access to the TA: ***danielmarin350@gmail.com***
  - Report modify the provided doc file
    - Provide your git repo link
    - Answer the question and put your code there
    - Submit PDF file to blackboard

# Announcement

- Homework 1
  - **Get the installation done this week**
    - We will do a in class exercise next week
    - Bring your laptop

# Acknowledgements

- “Operating Systems Concepts” book and supplementary material by A. Silberschatz, P. Galvin and G. Gagne
- “Operating Systems: Internals and Design Principles” book and supplementary material by W. Stallings
- “Modern Operating Systems” book and supplementary material by A. Tanenbaum
- R. Doursat and M. Yuksel from University of Nevada, Reno
- Farshad Ghanei from Illinois Tech
- T. Kosar and K. Dantu from University at Buffalo