Threads & Concurrency

DSC 315: Computer Organization & Operating Systems

Dr. Laltu Sardar

School of Data Science,
Indian Institute of Science Education and Research Thiruvananthapuram (IISER TVM)

March 18 & 23, 2026

Table of Content

1. Threads & Concurrency
   - Introduction
   - Multicore Programming
   - Multithreading Models
   - Multithreading Models
   - Threading Libraries
   - Threading Issues
   - Issues in Thread Cancellation

Recommended Reading From textbook:
Chapters 4.1, 4.3, 4.4: 4.4.1-4.4.2 4.5: 4.5.1, 4.6, 4.7
Mandatory to learn the codes discussed in the class.

IISER THIRUVANANTHAPURAM

4

Threads & Concurrency

Thread in Operating System

Definition

A **thread** is the smallest unit of execution within a process that can be scheduled by the CPU.

Properties

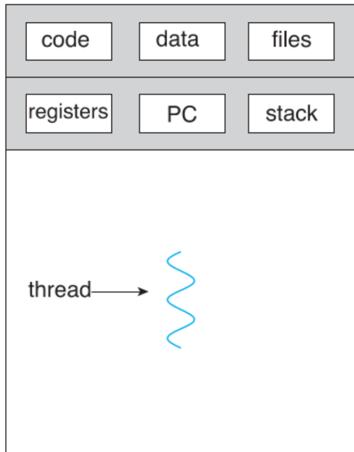
- Multiple threads within the same process execute independently while sharing the process resources.
- Thread state contains: Thread ID, Program Counter (PC), Register set, Stack
- Shared process resources: Code section, Data section, Open files, Signals, other OS resources

Note

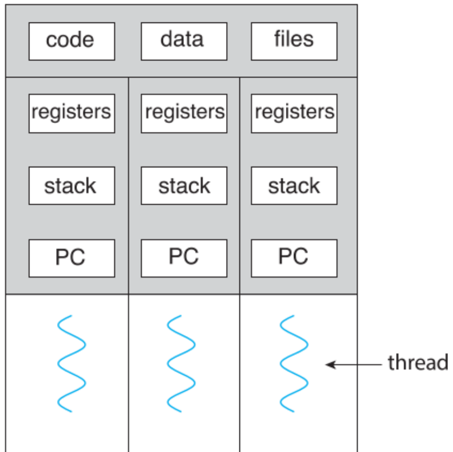
- A traditional process has a **single thread of control**.
- A **multithreaded process** contains multiple threads that can execute tasks concurrently.

IISER THIRUVANANTHAPURAM

Single-threaded vs multithreaded processes.



single-threaded process



multithreaded process

Process vs Thread

	Process	**Thread**
Control Block	Creates a Process Control Block (PCB) containing process information	Creates a Thread Control Block (TCB) containing thread information
State Information	Includes code section, data section, file info, registers, PC, stack	Contains only register set, program counter, and stack
Resource Ownership	Owns code, data, and file information independently	Shares code, data, and files with other threads in the process
Creation Cost	Higher overhead since all process resources are initialized	Lower overhead since only execution state is created

IISER THIRUVANANTHAPURAM

Kernel Threads in Linux

Kernel Threads

- During Linux system boot, several **kernel threads** are created by the kernel.
- These threads run in **kernel space** and support core operating system functions.
- Each kernel thread performs a specific task such as:
  - Device management
  - Memory management
  - Interrupt handling
  - Background system services
- Kernel threads do not execute user programs but assist the kernel in managing system resources.
- The command `ps -ef` can be used to view kernel threads on a running Linux system.

Note:

- The first threads created during system boot are **kernel threads**, which run only in kernel mode.
- **systemd** (PID 1) is the first process created in user mode.

IISER THIRUVANANTHAPURAM

Benefits of Multithreading

- **Responsiveness**
  - An interactive application can continue running even if part of it is blocked or performing a lengthy operation
  - This increases responsiveness to the user
- **Resource Sharing**
  - Threads share the memory and resources of the process to which they belong by default
  - Multiple threads can work within the same address space
- **Economy**
  - Allocating memory and resources for process creation is costly
  - Thread creation and context switching are therefore cheaper
- **Scalability**
  - Threads can run in parallel on multiple CPU cores
  - Improves performance in multiprocessor systems

IISER THIRUVANANTHAPURAM

Multicore Programming

Multicore Programming

Challenges in Multicore Systems

- Multicore or multiprocessor systems put additional pressure on programmers
- Key challenges include:
  - **Dividing activities into parallel tasks** Identify program parts that can run independently. Independent tasks can execute on different cores.
  - **Maintaining load balance among cores** Tasks should have roughly equal workloads. Otherwise some cores remain idle.
  - **Splitting data across tasks** Data must also be divided among tasks. Each core processes a portion of the data.
  - **Managing data dependencies** Tasks may depend on results from others. Synchronization is required to ensure correctness.
  - **Testing and debugging parallel programs** Parallel programs have many execution paths. Debugging is harder than single threaded programs.

Concurrency vs Parallelism

- **Concurrency:** A Concurrent system supports more than one task by allowing all tasks to make progress. **Concurrency** means that some threads may run in parallel
- **Parallelism:** A system can perform more than one task simultaneously

IISER THIRUVANANTHAPURAM

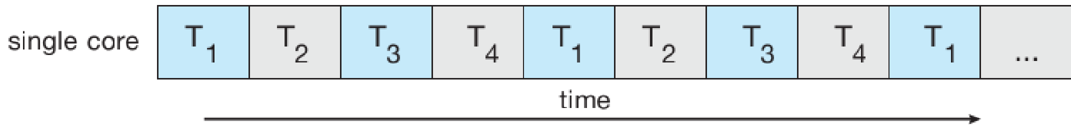


Figure: Concurrent execution on a single-core system.

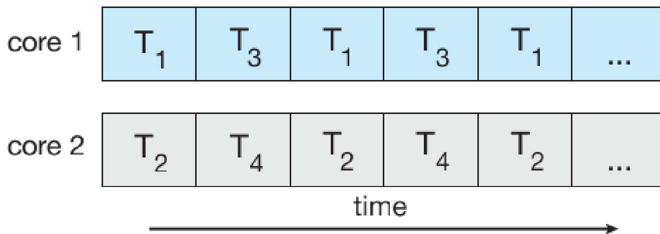
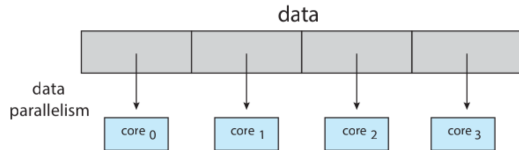


Figure: Parallel execution on a multicore system.

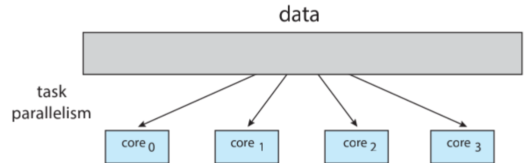
Types of Parallelism

Data Parallelism

- Same operation on different parts of data
- Data divided across multiple cores



Task Parallelism



- Different tasks executed in parallel
- Each thread performs a different function

Amdahl's Law

Idea

- Amdahl's Law estimates the performance gain when additional CPU cores are added to a program.
- It considers that a program contains both **serial** and **parallel** components.

Formula

$$\text{Speedup} = \frac{1}{S + \frac{1-S}{N}}$$

- S : Serial portion of the program
- N : Number of processing cores

Key Observations

- If a program is 75% parallel and 25% serial, using 2 cores gives about **1.6× speedup**.
- As $N \to \infty$, the maximum speedup approaches $1/S$.
- Even a small serial portion can limit performance improvement.

IISER THIRUVANANTHAPURAM

Multithreading Models

Multithreading Models

User Threads vs Kernel Threads

- **User Threads**
  - Managed by user level thread libraries
  - Kernel is unaware of these threads
  - Thread management is fast
- **Kernel Threads**
  - Managed by the operating system kernel
  - Kernel schedules and manages threads
  - Supports true parallel execution

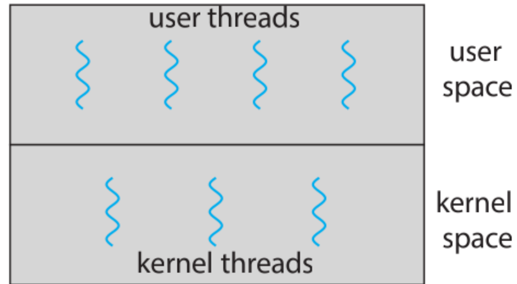
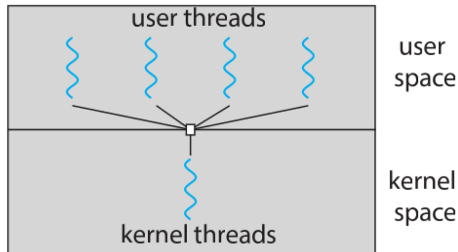


Figure: User and kernel threads.

Multithreading Models (1)

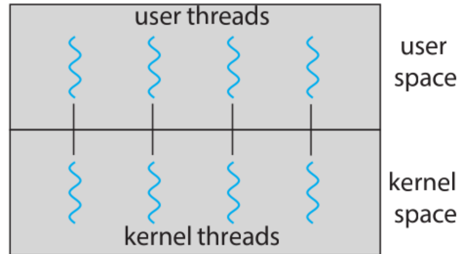
Many-to-One Model

- Many user threads mapped to one kernel thread
- Thread management in user space
- If one thread blocks, entire process blocks



One-to-One Model

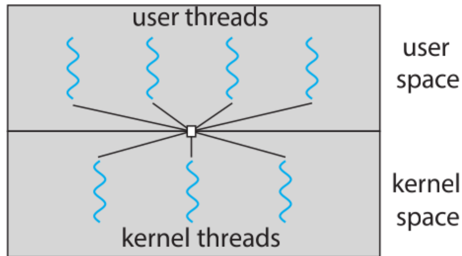
- Each user thread mapped to a kernel thread
- Supports parallel execution
- Higher overhead for thread creation



Multithreading Models (2)

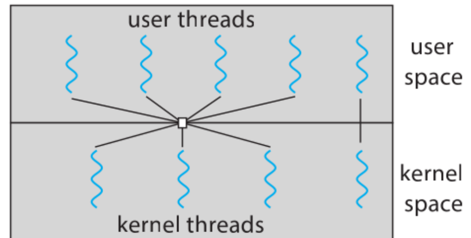
Many-to-Many Model

- Many user threads mapped to multiple kernel threads
- Kernel schedules kernel threads on processors
- Reduces overhead of kernel thread creation



Two-Level Model

- Variation of many-to-many model
- Some user threads bound to specific kernel threads
- Provides more scheduling control



Threading Libraries

Thread Libraries

Thread Library

- A **thread library** provides an API for creating and managing threads in a program.
- It allows programmers to perform operations such as:
  - Creating threads
  - Terminating threads
  - Synchronizing threads
  - Scheduling thread execution
- Thread libraries may be implemented in **user space** or supported by the **kernel**.
- Common thread libraries include:
  - POSIX Threads (Pthreads)
  - Windows Threads
  - Java Threads

IISER THIRUVANANTHAPURAM

Synchronous vs Asynchronous Threading

Asynchronous Threading

- Parent creates a child thread and continues execution immediately
- Parent and child run independently and concurrently
- Usually involves little data sharing
- Common in multithreaded servers and responsive user interfaces

Synchronous Threading

- Parent creates child threads and waits for them to finish
- Parent resumes only after all children terminate
- Threads often share and combine results
- Typically uses operations such as thread `join()`

IISER THIRUVANANTHAPURAM

Examples

Example widely used threads are

1. Pthreads: `Code with Single Thread`, `Code with Multiple Threads`
2. Windows Thread: `Link to the code`
3. Java Thread:

Similarities

- Both provide APIs for creating and managing threads
- Support thread synchronization (mutex, locks, etc.)
- Allow concurrent execution within a process
- Used for multithreaded programming

Differences

- **Pthreads:** POSIX standard API used in UNIX systems
- **Windows Threads:** Native API of Windows OS
- **Pthreads:** Specification based, implementation may vary
- **Windows Threads:** Directly implemented by Windows kernel

IISER THIRUVANANTHAPURAM

Pthreads Example: Program Structure

`Example Code for creating a single thread`

- All Pthreads programs include the header file `pthread.h`.
- `pthread_t tid` declares the identifier of the thread to be created.
- Each thread has attributes such as stack size and scheduling information.
- `pthread_attr_t attr` represents the thread attributes.
- `pthread_attr_init(&attr)` initializes the attributes with default values.
- A new thread is created using `pthread_create()`.
- The function name `runner()` specifies where the new thread begins execution.

IISER THIRUVANANTHAPURAM

Pthreads Example: Execution Flow

- After thread creation, two threads exist:
  - Parent thread executing `main()`
  - Child thread executing `runner()`
- The child thread computes the summation using the parameter from `argv[1]`.
- The parent thread waits for the child using `pthread_join()`.
- The child thread terminates using `pthread_exit()`.
- After the child finishes, the parent prints the shared variable `sum`.
- This example demonstrates creation and synchronization of a **single thread**.

Pthreads Example: Multiple Thread Creation

Example Code for creating a Multiple threads

- In multicore systems, programs often create **multiple threads** to perform tasks concurrently.
- Instead of managing each thread separately, threads can be stored in an array:
  - `pthread_t workers[NUM_THREADS]`
- A loop can be used to create several threads:
  - `pthread_create(&workers[i], &attr, runner, param)`
- The parent thread can wait for all threads using a **join loop**:
  - `for (i = 0; i < NUM_THREADS; i++)`
  - `pthread_join(workers[i], NULL);`
- This approach simplifies managing many threads and is common in parallel programs.

Implicit Threading

- Managing many threads manually can be complex for programmers.
- **Implicit threading** allows the compiler or runtime system to manage thread creation and scheduling automatically.
- Programmers specify the **tasks**, while the system decides how threads are created and executed.
- Benefits:
  - Simplifies parallel programming
  - Improves scalability on multicore systems
  - Reduces programmer burden
- Examples include parallel libraries such as OpenMP and Grand Central Dispatch.

IISER THIRUVANANTHAPURAM

Thread Pool

- Creating and destroying threads repeatedly is expensive.
- A **thread pool** maintains a set of precreated worker threads.
- Tasks are placed in a queue and assigned to available threads in the pool.
- Advantages:
  - Reduces overhead of thread creation
  - Limits the number of concurrent threads
  - Improves performance in server applications

IISER THIRUVANANTHAPURAM

Example of Thread Pool

- A web server handling many client requests uses a thread pool.
- The server creates a fixed number of worker threads when it starts.
- When a client request arrives:
  - The request is placed in a task queue
  - An available thread from the pool processes the request
- After finishing, the thread returns to the pool and waits for the next task.
- This approach improves responsiveness and system efficiency.

IISER THIRUVANANTHAPURAM

Threading Issues

Threading Issues

- Multithreaded programs introduce several design and implementation challenges.
- Important threading issues include:
  - Semantics of `fork()` and `exec()` system calls
  - Signal handling in multithreaded processes
  - Thread cancellation
  - Thread-local storage
  - Scheduler activations
- These issues arise because multiple threads execute within the same process and share resources.

Threading Issues: `fork()` and `exec()`

- In a multithreaded process, the behavior of `fork()` raises an important question: Does the new process duplicate **all threads** or only the **calling thread**?
- Some UNIX systems provide two versions of `fork()`:
  - Duplicate only the thread that invoked `fork()`
  - Duplicate all threads of the process
- The `exec()` system call replaces the entire process image, including all threads.
- If `exec()` is called immediately after `fork()`, duplicating only the calling thread is sufficient.
- If the child process does **not** call `exec()`, it should duplicate all threads.

IISER THIRUVANANTHAPURAM

Signal Handling in Multithreaded Systems

- Signals in UNIX notify a process that a particular event has occurred.
- A **signal handler** processes the signal:
  1. Signal is generated by an event
  2. Signal is delivered to a process
  3. Signal is handled by a handler
- Types of signal handlers:
  - **Default handler** provided by the kernel
  - **User defined handler** that overrides the default
- In a **single threaded process**, the signal is delivered to the process.
- In a **multithreaded process**, signals may be handled by:
  - The thread to which the signal applies
  - Every thread in the process
  - Selected threads in the process
  - A dedicated thread assigned to receive all signals

IISER THIRUVANANTHAPURAM

Signal Handling in Multithreaded Systems (Cont.)

- In multithreaded processes, signals can be delivered in several ways:
  - To the thread to which the signal applies
  - To every thread in the process
  - To selected threads in the process
  - To a dedicated thread that handles all signals
- **Synchronous signals** (e.g., divide-by-zero) are delivered to the thread that caused the signal.
- **Asynchronous signals** (e.g., `Ctrl+C`) may be delivered to multiple threads.
- UNIX provides the function:
  - `kill(pid_t pid, int signal)` to send a signal to a process
- POSIX Pthreads also provides:
  - `pthread_kill(pthread_t tid, int signal)` to send a signal to a specific thread
- Typically, the signal is delivered to the first thread that is not blocking it.

IISER THIRUVANANTHAPURAM

Signal Handling in Windows

- Windows does not provide UNIX-style signals.
- Instead, it uses **Asynchronous Procedure Calls (APCs)**.
- A thread registers a function that executes when a specific event occurs.
- APCs are delivered to a **specific thread**, not the entire process.
- This makes signal-like handling simpler in multithreaded programs.

Issues in Thread Cancellation

Thread Cancellation

- **Thread cancellation** means terminating a thread before it finishes execution.
- The thread being terminated is called the **target thread**.
- Two general approaches:
  - **Asynchronous cancellation**: target thread is terminated immediately
  - **Deferred cancellation**: thread periodically checks if it should terminate
- Cancellation is difficult when the thread holds resources or updates shared data.
- Asynchronous cancellation may leave resources unreleased.

IISER THIRUVANANTHAPURAM

Thread Cancellation Modes and API

Cancellation Modes

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

Important Pthreads API

Function	Purpose
`pthread_cancel(tid)`	Send cancellation request
`pthread_setcancelstate()`	Enable/disable cancellation
`pthread_setcanceltype()`	Set deferred or async type
`pthread_testcancel()`	Check cancellation point

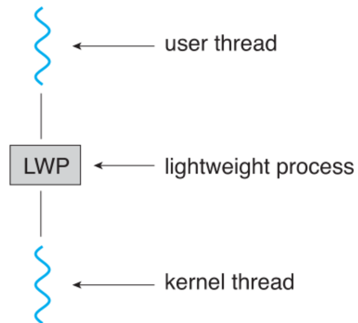
IISER THIRUVANANTHAPURAM

Thread Cancellation in Pthreads

- Pthreads supports several **cancellation modes**.
- If cancellation is disabled, a thread cannot be canceled.
- In **deferred cancellation**, termination occurs only at defined **cancellation points**.
- Function `pthread_testcancel()` checks if a cancellation request exists.
- If a request is pending, the thread terminates; otherwise execution continues.
- Pthreads also supports **cleanup handlers** to release resources when a thread is canceled.

Scheduler Activations

- In **Many-to-Many** and **Two-Level** models, coordination is required between user threads and kernel threads.
- An intermediate structure called a **Lightweight Process (LWP)** is used.
- LWP behaves like a **virtual processor** on which user threads run.
    - Each LWP is attached to a kernel thread
    - User threads are scheduled onto LWPs
- A key question is **how many LWPs** should be allocated to a process.
- **Scheduler activations** provide communication between the kernel and the thread library.
- The kernel sends **upcalls** to the thread library when events occur.
- This mechanism helps maintain the appropriate number of kernel threads for the application.



IISER THIRUVANANTHAPURAM

References

Textbook:

1. Avi Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Global Edition. Wiley, 2023. ISBN: 9781119320913.

Slides:

1. Avi Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts, Tenth Edition: Lecture Slides*. Wiley, 2018. Available at: `https://os-book.com/OS10/slide-dir/index.html`. Accessed: 2026-03-06.

IISER THIRUVANANTHAPURAM

Dr. Laltu Sardar, Assistant Professor, IISER Thiruvananthapuram
laltu.sardar@iisertvm.ac.in, laltu.sardar.crypto@gmail.com
Course webpage: https://laltu-sardar.github.io/courses/corgos_2026.html.