**Chapter 04: Threads**

Table of Contents

[4.1 Overview 2](#_Toc83032634)

[Advantages 3](#_Toc83032635)

[4.2 Multicore Programming 4](#_Toc83032636)

[Amdahl’s Law 4](#_Toc83032637)

[Programming Challenges 5](#_Toc83032638)

[4.3 Multithreading Models 7](#_Toc83032639)

[Many-to-One Model 7](#_Toc83032640)

[One-to-One Model 7](#_Toc83032641)

[Many-to-Many Model 8](#_Toc83032642)

[4.5 Implicit Threading 10](#_Toc83032643)

[Thread Pools 10](#_Toc83032644)

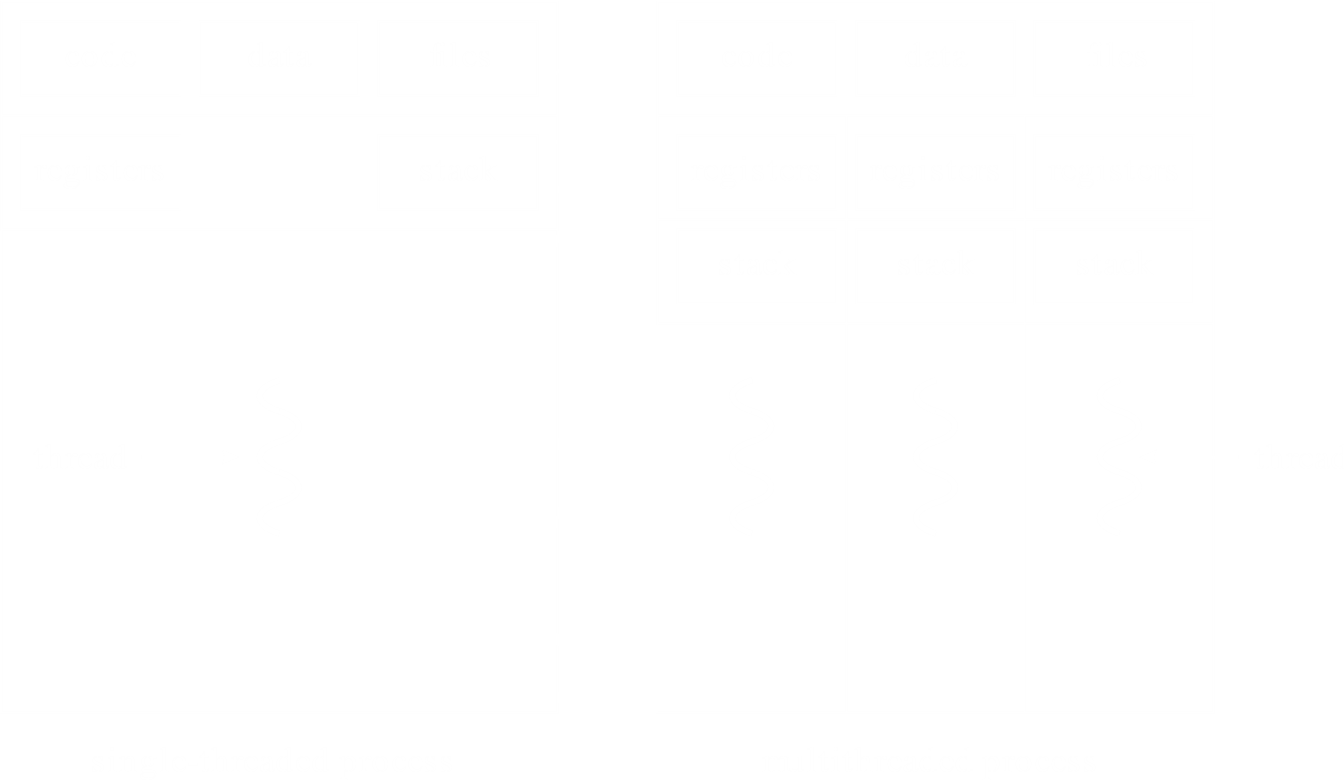
[4.6 Threading Issues 11](#_Toc83032645)

[Thread Cancellation 11](#_Toc83032646)

## 4.1 Overview

Creating a **process** is both time consuming and resource intensive. Because of this, generally, **threads** are created. A thread is an **execution unit** of a process.

Say a client requests a server for a particular service. To provide this service, the server does not create a new process. The server already knows what possible services could be requested, so it already has the processes created. When the client requests the service, the server creates a thread for the required process and assigns it for use to the client.



The situation will look a little like the image above. We have a single process in both cases, but for the diagram on the left, only a single thread is running, whereas for the diagram on the right, multiple threads for the same process, using the same metadata, are running.

### Advantages

**Multithreading**, the process of creating multiple threads of a process, reduces the resource intensiveness of process creation. All the **resources** given to the parent process are **shared** amongst the threads.

This causes:

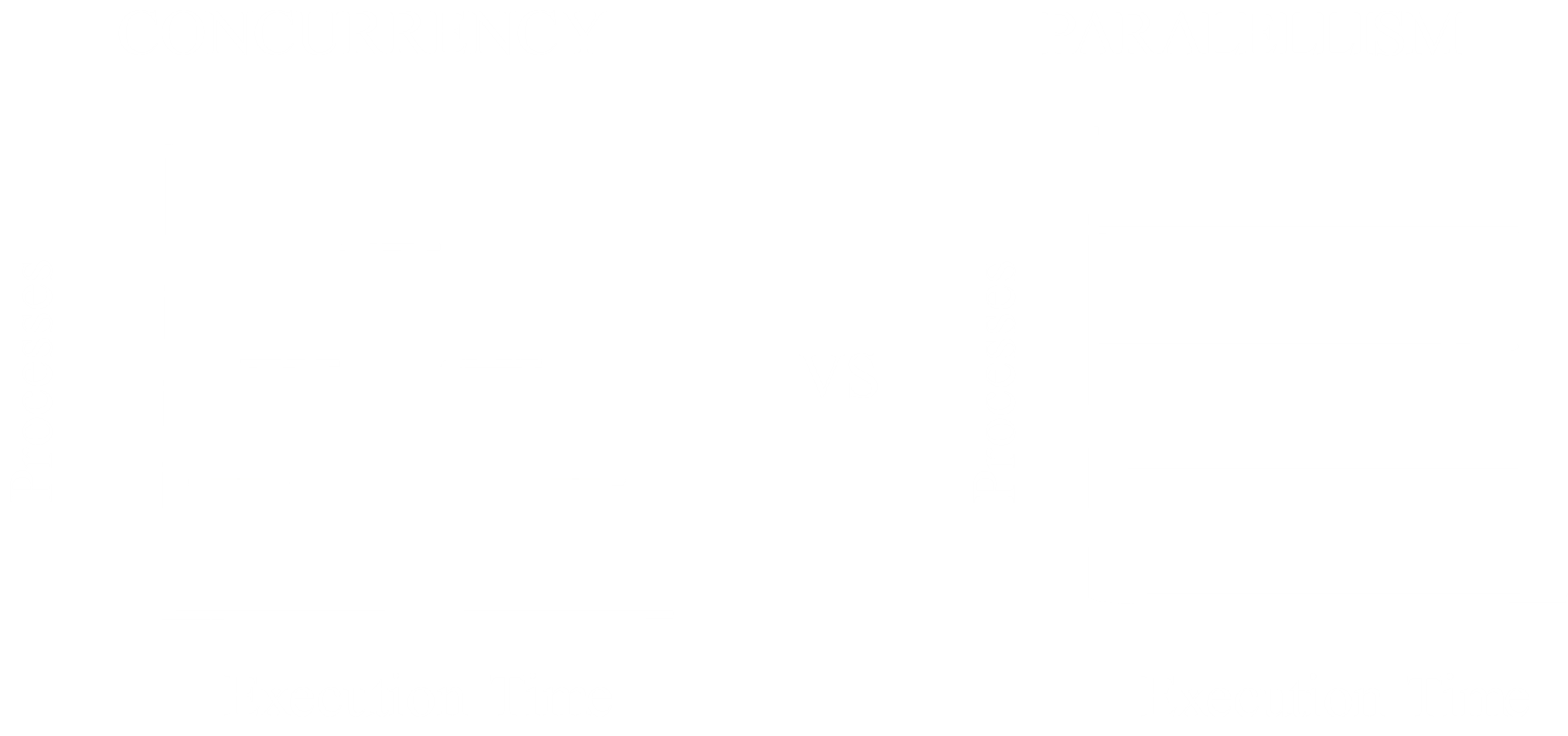
* **Responsiveness** – Multithreaded applications may allow a program to continue running even if a part of it is blocked or performing a lengthy operation. This seemingly increases responsiveness for the end-user.
* **Resource Sharing** – Since this happens by default, it is more convenient that resource sharing amongst processes, which must be explicitly programmed.
* **Scalability** – A single-threaded process can only run on one processor. Multithreading allows us to take advantage of multiprocessor architecture.
* **Economy** – The time saved and the resource sharing makes multithreading more economical than creating separate processes.

## 4.2 Multicore Programming

A **parallel** system is one that allows more than one task to run simultaneously.

A **concurrent** system is one that allows more than one task by allowing all the tasks to make progress.

The difference here is, **concurrency** is when multiple tasks can have **overlapping time periods** but are not actually running at the **same time**. For example, one task could start, pause, allow a different task to start, pause the second task and start the first one again. By contrast, **parallelism** is when multiple tasks are literally running **simultaneously**.



### Amdahl’s Law

Say we have an application that has one part that must be executed **serially** and another part that can be executed **parallelly**. If we consider to be the portion of the application that must be executed serially, then the portion that can be executed parallelly is .

In a system with cores, using the cores for the portion that can be executed parallelly leads to the following **speedup**:

For example, if , then having cores gets us a speedup of times, while having cores gets us a speedup of times.

This is **Amdahl’s Law**.

As is increased, the speedup converges to . We cannot have a speedup more than , no matter how many cores we add. This is the fundamental principle behind the law. The **serial portion** of an application can have a **disproportionate effect** on the performance gains.

### Programming Challenges

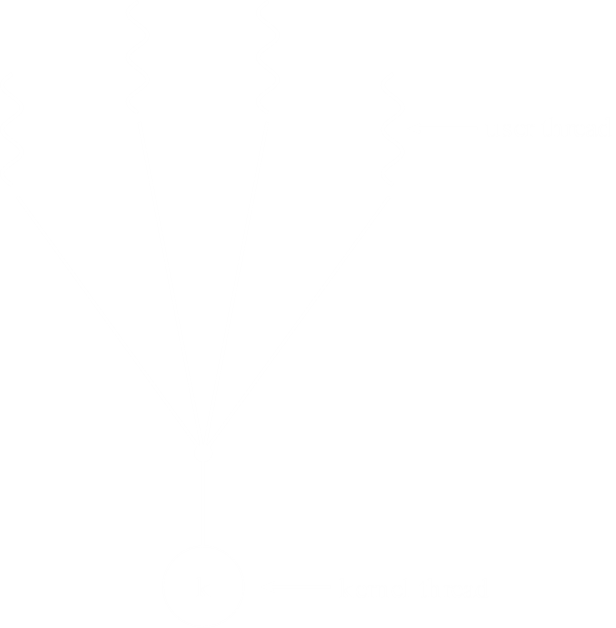
In general, five areas present challenges in programming for multicore systems:

1. **Identifying Tasks** – We must examine applications and identify areas that can be divided into separate, concurrent tasks. Tasks are ideally independent of one another.
2. **Balance** – We also need to ensure that parallel tasks perform equal work of equal value. If one task does not contribute much to the overall process, assigning it a separate execution core might not be worth the cost.
3. **Data Splitting** – The data accessed by the tasks must also be divided to run on separate cores.
4. **Data Dependency** – When one task depends on the data from another, we need to ensure the tasks are synchronized to accommodate the dependency.
5. **Testing and Debugging** – When a program is running in parallel on multiple cores, many different execution paths are possible. Testing and debugging concurrent programs is thus more difficult than it would be for single-threaded applications.

## 4.3 Multithreading Models

### Many-to-One Model

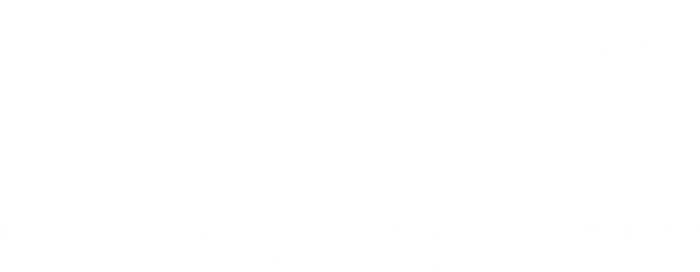
In the **many-to-one model**, we have a **single kernel thread** from which we create **multiple user threads**.



This model is **efficient** to some extent. However, if a single user thread makes a **blocking system call**, the entire process will be blocked. Only **one thread** can access the kernel at a time, which means we cannot have parallelism.

### One-to-One Model

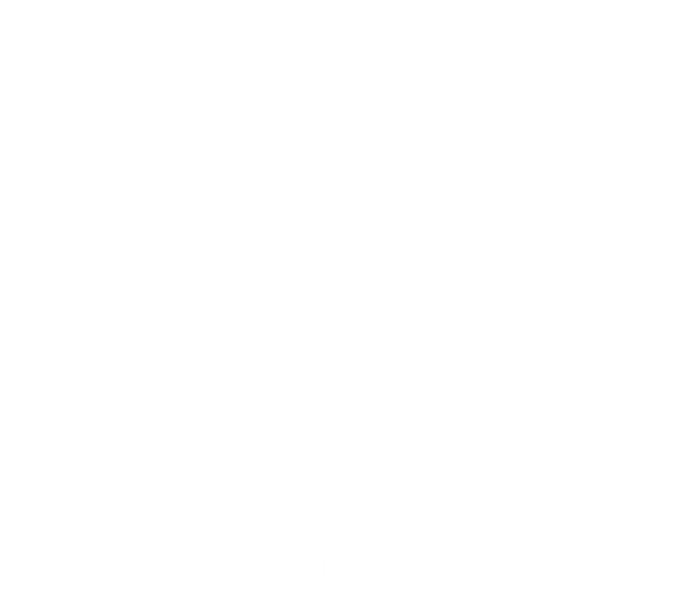
In the **one-to-one** model, we have **multiple kernel thread**, each with a **single user thread**.



Since every user thread is attached to a different kernel thread, the **blocking system call** issue is dealt with. However, there is **overhead** due to the fact that we need to create a kernel thread for every user thread we create. Because of this, there is usually a **limit** to the number of threads we can create.

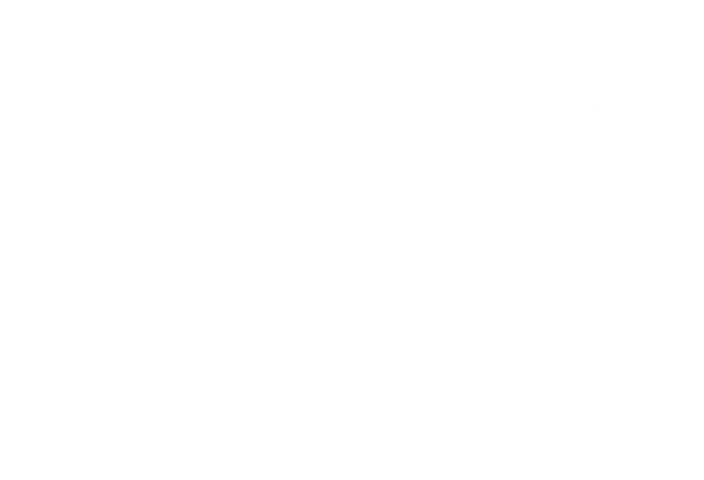
### Many-to-Many Model

In the **many-to-many model**, we have **multiple kernel threads** and **multiple user threads**. However, we do not necessarily know which user thread is connected to which kernel thread.



Since multiple user threads are accessing a **smaller or equal number** or kernel threads, **multiplexing** is involved. There is some form of **hub** in between. This model overcomes both the **blocking system call** issue and the **overhead** issue of the previous models.

A variation of the many-to-many model is called the **two-level model**.



## 4.5 Implicit Threading

### Thread Pools

The concept of a **thread pool** is that a number of threads are created when a process **first starts**. By doing this, whenever a request is received, a thread from the pool, if available, can be **awakened** and **assigned** to work. Once the work is completed, the thread can be returned to the pool.

The benefits of using thread pools are:

* It is **faster** than waiting to create a thread.
* It **limits the number of threads** that can be created, which can be beneficial for weaker systems.
* It separates the actual task from the mechanics of creating a task. A task simply comes and takes a thread. This allows us to use **different strategies** for running the task.

## 4.6 Threading Issues

### Thread Cancellation

**Thread cancellation** occurs when a thread is terminated **before it completes** its work. For example, if multiple threads are searching a database and one thread finds the requested results, the other threads can be cancelled.

A thread that is to be cancelled is called the **target thread**.

Thread cancellation can occur it two ways:

* **Asynchronous** – One thread immediately terminates the target thread.
* **Deferred** – The target thread terminates itself by periodically checking if it needs to terminate.