**Chapter 4**

**Threads**

This passage discusses threads, which are a fundamental concept in operating systems and concurrent programming. Let's break down the key points:

\*\*Definition of a Thread:\*\*

- A thread is a basic unit of CPU utilization.

- It consists of several components, including a thread ID, a program counter (PC), a register set, and a stack.

- Threads within the same process share common resources, including the code section, data section, and other OS resources like open files and signals.

- Unlike traditional heavyweight processes, which have a single thread of control, multithreaded processes have multiple threads that can perform tasks concurrently.

\*\*Motivation for Multithreading:\*\*

- Many modern software applications are multithreaded, meaning they are implemented as separate processes with multiple threads of control.

- Multithreading allows applications to perform multiple tasks concurrently. For example, a web browser may use one thread to display content and another to fetch data from the network.

- Multithreading is crucial for efficiently utilizing the processing power of multi-core systems.

- Multithreading is used in scenarios where multiple similar tasks need to be performed. Instead of creating a new process for each task, threads within the same process can handle them more efficiently.

\*\*Benefits of Multithreading:\*\*

1. \*\*Responsiveness:\*\* Multithreading can make interactive applications more responsive. Even if one thread is blocked or performing a lengthy operation, other threads can continue running, keeping the application responsive to user input.

2. \*\*Resource Sharing:\*\* Threads within the same process share resources by default, making it easier for them to communicate and share data compared to processes, which require explicit mechanisms like shared memory or message passing.

3. \*\*Economy:\*\* Creating and managing threads is generally more efficient than creating and managing processes. Threads within the same process share memory and resources, reducing the overhead of resource allocation.

4. \*\*Scalability:\*\* Multithreading becomes even more valuable in multiprocessor systems, where threads can run in parallel on different CPU cores. This allows for efficient utilization of multiple processing cores.

\*\*Concurrent Execution:\*\*

- Threads can execute concurrently, which means they can run simultaneously or in an interleaved manner, depending on the available hardware resources.

- On a single-core system, multiple threads are scheduled to run in a time-sharing manner, as shown in Figure 4.3.

- On multi-core systems, threads can run truly in parallel, taking advantage of the available CPU cores.

In summary, threads are a fundamental concept in concurrent programming and are widely used in modern software applications and operating systems to improve responsiveness, resource sharing, efficiency, and scalability. They enable multiple threads of control within a single process, allowing for efficient multitasking and utilization of hardware resources.

**Types of Parallelism**

\*\*Data Parallelism:\*\*

- Data parallelism focuses on dividing a large dataset into smaller subsets and processing each subset concurrently on different computing cores or processors.

- The same operation is applied to each subset of the data independently and in parallel.

- It is particularly effective when you have a large amount of similar data to process, and the same computation needs to be performed on each data element.

- An example of data parallelism is parallelizing the task of summing the elements of an array. Each core or thread can independently sum a portion of the array, and the results can be combined later.

- In data parallelism, multiple threads work on different portions of the data simultaneously.

\*\*Task Parallelism:\*\*

- Task parallelism involves dividing a complex task or problem into smaller, independent tasks or threads that can be executed in parallel on separate computing cores or processors.

- Each thread performs a unique and potentially different operation or task.

- Task parallelism is useful when different operations need to be performed concurrently, and these operations may or may not work on the same data.

- An example of task parallelism is running two threads, where one thread performs statistical analysis on a dataset, and another thread generates graphical visualizations of the same data. Both threads operate independently on their respective tasks.

- In task parallelism, multiple threads perform different tasks concurrently.

\*\*Hybrid Approach:\*\*

- In practice, many applications use a combination of both data and task parallelism, depending on the nature of the problem and the available hardware resources.

- For example, a scientific simulation may involve data parallelism for performing calculations on large datasets and task parallelism for handling various aspects of the simulation, such as data visualization, input/output operations, and communication between different simulation components.

- The goal is to efficiently utilize the available computing resources and improve overall performance by parallelizing both data processing and task execution.

In summary, data parallelism focuses on parallelizing the processing of similar data across multiple cores, while task parallelism involves parallelizing independent tasks across cores. Many real-world applications use a combination of these two approaches to effectively harness the power of parallel computing and achieve better performance. The choice between data and task parallelism depends on the specific requirements and characteristics of the application and the underlying hardware architecture.

**Multithreading Models**

\*\*Many-to-One Model:\*\*

- In the many-to-one model, there are many user-level threads (threads managed by the application) mapped to just one kernel thread (managed by the operating system).

- The advantage is that it's efficient because thread management is done by the application itself.

- However, the big downside is that if one thread makes a blocking operation (like waiting for input), the entire process (including all threads) can be blocked. This is not suitable for multi-core processors because only one thread can run at a time.

\*\*One-to-One Model:\*\*

- In the one-to-one model, each user-level thread is directly mapped to a kernel thread.

- This model allows for more concurrency because when one user-level thread blocks, others can continue running. It's suitable for multi-core processors.

- However, creating a user-level thread requires creating a corresponding kernel thread, and this can have some overhead. Some systems limit the number of threads to manage this overhead.

\*\*Many-to-Many Model:\*\*

- The many-to-many model is a hybrid approach that combines the benefits of both the many-to-one and one-to-one models.

- Many user-level threads are mapped to a smaller number of kernel threads. The exact mapping can vary based on the application or the system's configuration.

- This model allows developers to create as many user-level threads as needed, and these threads can run in parallel on multi-core processors. When one user-level thread blocks, others can still run.

- Some variations of this model allow user-level threads to be explicitly bound to kernel threads for better control.

In simpler terms, these models determine how user-level threads (threads created and managed by your program) are connected to kernel threads (threads managed by the operating system). The goal is to balance efficiency and concurrency so that your program can make the most of your computer's processing power, especially on multi-core machines.

**Thread Pools**

A thread pool is a technique used in multithreaded programming to efficiently manage threads and execute tasks concurrently. It addresses common issues associated with creating and managing threads in applications like web servers, where many client requests need to be handled simultaneously. Here's a simplified explanation:

1. \*\*Creating Threads Takes Time:\*\* In a multithreaded program, creating a new thread for each incoming request can be slow. Threads have some overhead, and creating and destroying them frequently is inefficient.

2. \*\*Resource Management:\*\* Allowing an unlimited number of threads to be created can lead to resource exhaustion. Too many threads may consume excessive CPU time and memory.

\*\*Thread Pool to the Rescue:\*\*

A thread pool solves these problems by pre-creating a pool of threads at the program's start. These threads sit ready and waiting for tasks to be assigned to them. When a new request arrives, instead of creating a new thread, the program assigns the task to one of the existing idle threads from the pool. Once the task is completed, the thread returns to the pool, waiting for more work.

\*\*Benefits of Thread Pools:\*\*

- \*\*Faster Task Execution:\*\* Using an existing thread from the pool is quicker than creating a new one from scratch.

- \*\*Controlled Resource Usage:\*\* Thread pools limit the number of threads that can exist simultaneously, preventing resource exhaustion.

- \*\*Flexibility:\*\* Thread pools allow for various task scheduling strategies, such as running tasks with a time delay or periodically.

\*\*Dynamic Thread Pool Sizing:\*\*

Some advanced thread pool designs can dynamically adjust the number of threads in the pool based on usage patterns. For example, they might reduce the number of threads when the system load is low, saving memory.

\*\*Examples:\*\*

- In Windows, the Thread Pool API provides functions like `QueueUserWorkItem()` to execute functions in a thread pool.

- In Java, the `java.util.concurrent` package includes utilities for creating and managing thread pools.

Overall, thread pools are a valuable tool for efficient multithreading, improving application performance, and managing system resources effectively.

**Threading Issues**

**The fork() and exec() System Calls**

In a multithreaded program, the behavior of the `fork()` and `exec()` system calls, which are commonly used in UNIX-like operating systems, can be a bit more complex than in single-threaded programs. Let's break down how these system calls work in the context of multithreaded programs:

1. \*\*fork() System Call:\*\*

- In a single-threaded program, when you call `fork()`, it creates a duplicate of the entire process, including the parent process's memory, code, and data. The new child process is essentially a clone of the parent.

- In a multithreaded program, the behavior of `fork()` depends on the operating system and the specific version of the C library being used.

- Some UNIX systems provide two versions of `fork()`:

- \*\*"Full" fork:\*\* This version duplicates all threads in the parent process. The child process will have the same set of threads as the parent.

- \*\*"Single-threaded" fork:\*\* This version duplicates only the thread that invoked the `fork()` system call. The child process starts as a single-threaded process and doesn't inherit the other threads from the parent.

- Which version of `fork()` to use depends on your application's requirements. If you intend to call `exec()` immediately after `fork()` in the child process, you may prefer the "single-threaded" fork. This is because the child process will replace its entire memory space with a new program when `exec()` is called, so duplicating all threads is unnecessary.

- If the separate process created by `fork()` does not call `exec()` and you want the child process to continue running with all threads intact, then you would use the "full" fork.

2. \*\*exec() System Call:\*\*

- The `exec()` system call is used to replace the current process's code and data with a new program. In a multithreaded program, when a thread invokes `exec()`, the entire process, including all threads, is replaced by the new program specified in the `exec()` call.

In summary, whether you use "full" or "single-threaded" `fork()` in a multithreaded program depends on whether you intend to replace the entire process (including all threads) with a new program using `exec()` immediately after the `fork()`. If you plan to execute a new program with `exec()`, using "single-threaded" fork is more efficient, as it doesn't duplicate unnecessary threads in the child process. If you want the child process to continue with all threads, you should use "full" fork.

**Signal Handling**

In UNIX systems, signals are used to notify processes about specific events or conditions. Signals can be categorized into two main types: synchronous signals and asynchronous signals.

\*\*Synchronous Signals:\*\*

- Synchronous signals are generated as a direct result of an action or event that occurs within a running process. For example, if a process attempts an illegal memory access or divides by zero, a synchronous signal is generated.

- These signals are delivered to the same process that triggered the event, and they are considered "synchronous" because they are directly tied to the process's actions.

- When a synchronous signal is generated, it interrupts the normal execution of the process and typically results in the termination of the process if not handled.

\*\*Asynchronous Signals:\*\*

- Asynchronous signals, on the other hand, are generated externally to a process and are not directly tied to the actions of the process.

- These signals are sent to a process from an external source, such as user actions like pressing Ctrl+C to terminate a program or a timer expiring.

- Asynchronous signals are "asynchronous" because they can arrive at any time, independent of the process's current activities.

\*\*Handling Signals in Single-Threaded Programs:\*\*

In single-threaded programs, handling signals is relatively straightforward. When a signal is received, it is delivered to the entire process. The process can have a default action for handling the signal, which may include terminating the process, or it can specify a custom (user-defined) signal handler function to handle the signal's specific action.

\*\*Handling Signals in Multithreaded Programs:\*\*

Handling signals in multithreaded programs is more complex because there are multiple threads within a single process. Here are some considerations:

1. \*\*Delivering Signals:\*\* When a signal is generated, the question arises as to which thread should receive and handle the signal. There are several options:

- Deliver the signal to the specific thread that triggered the event (for synchronous signals).

- Deliver the signal to all threads in the process (for certain asynchronous signals like program termination signals).

- Deliver the signal to specific threads based on criteria.

- Designate a specific thread to handle all signals for the process.

2. \*\*Thread-Specific Signal Handling:\*\* In many multithreaded systems, each thread can specify which signals it is willing to accept and which signals it wants to block. This allows for fine-grained control over signal handling at the thread level.

3. \*\*Functionality on Windows:\*\* Windows doesn't have signals in the same way as UNIX-like systems. Instead, it uses asynchronous procedure calls (APCs) to achieve similar functionality. An APC is a function that can be scheduled to run asynchronously in a specific thread.

In summary, handling signals in multithreaded programs involves deciding how to deliver signals to threads and ensuring that threads can control which signals they respond to. This complexity arises from the need to coordinate signal handling among multiple threads within a single process.

**Thread Cancellation**

Thread cancellation is a mechanism in multithreading that allows one thread to terminate another thread before it completes its normal execution. This feature is useful in scenarios where you want to stop a thread's execution for various reasons, such as completing a task earlier, responding to user actions, or dealing with exceptional situations. There are two main approaches to thread cancellation: asynchronous and deferred cancellation.

**Asynchronous Cancellation:**

In asynchronous cancellation, one thread (the requesting thread) immediately terminates the target thread (the thread to be canceled) without allowing it to perform any additional cleanup.

Asynchronous cancellation can be abrupt and may not give the target thread a chance to release any resources it has acquired or complete any critical operations safely.

This approach is generally discouraged because it can lead to resource leaks and unexpected behavior.

**Deferred Cancellation:**

Deferred cancellation is a more controlled approach to thread cancellation.

In deferred cancellation, the requesting thread signals its intention to cancel the target thread, but the actual cancellation occurs when the target thread reaches a predefined cancellation point.

The target thread periodically checks for cancellation requests and decides whether it should terminate itself gracefully.

Deferred cancellation provides a way for the target thread to release resources and perform necessary cleanup operations before termination, making it a safer option.