**CHAPTER-4**

**MULTITHREAD PROGRAMMING**

**4.1 Overview**

**Thread**: A thread is a basic unit of CPU utilization. It can be described as a single sequential flow of control within a program. It comprises of a thread ID, a program counter, a register set and a stack.

**4.1.2 Benefits**

1. *Responsiveness*: Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation.
2. *Resource Sharing*: Processes can only share resources through techniques such as shared memory and message passing, which have to be explicitly arranged by the programmer. However, threads share the memory and the resources of the process to which they belong by default.
3. *Economy*: Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads.
4. *Scalability*: Multithreading on a multiprocessor architecture helps threads to run in parallel on different processing cores.

**4.2 Multicore Programming**

**Multicore / Multiprocessor Systems**: Systems wherein multiple computing cores are placed on a single chip, and each core appears as a separate processor to the operating system, are known as multicore / multiprocessor systems.

*Distinction between Parallelism and Concurrency*: A system is parallel if it can perform more than one task simultaneously. In contrast, a concurrent system supports more than one task by allowing all tasks to make progress. Earlier, CPU schedulers provided the illusion of parallelism by rapidly switching between processes in the system, thereby allowing each process to make progress. Such processes were running concurrently, but not in parallel.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, application, table

Description automatically generated

**4.2.1 Programming Challenges**

1. *Identifying Tasks*: This involves examining applications to find areas that can be divided into separate, concurrent tasks.
2. *Balance*: Programmers must ensure that tasks perform work of equal value. In some instances, a certain task may not contribute as much value to the overall process as other tasks. Using a separate execution core to run that task may not be worth the cost.
3. *Data splitting*: Data accessed and manipulated by tasks must be divided to run on separate cores.
4. *Data dependency*: The data accessed by the tasks must be examined for dependencies between two or more tasks. When one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency.
5. *Testing and Debugging*: When a program is running in parallel on multiple cores, many different execution paths are possible. Testing and debugging such concurrent programs is inherently more difficult.

**4.2.2 Types of Parallelism**

1. *Data Parallelism*: Data Parallelism focusses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core.

For example, to sum the elements of an array, if a single-core system is used, one thread would sum all the contents. But on a dual-core system, the first half would be summed by thread A, while the other half would be summed by thread B.

1. *Task Parallelism*: Task Parallelism involves distributing data but not tasks (threads) across multiple containing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or on different data.

**4.3 Multithreading Models**

|  |  |
| --- | --- |
| *User Threads* | *Kernel Threads* |
| Support for user threads is provided at the user level itself. | Support for kernel threads is provided by the kernel. |
| User threads are supported above the kernel and are managed without kernel support. | Kernel threads are supported and managed directly by the operating system. |
| Benefits: No kernel modification, flexibility, and low cost. | Benefits: scheduling synchronization coordination, less overhead than process, suitable for parallel application. |
| Drawbacks: Thread may block entire process, no parallelism. | Drawbacks: More expensive than user-level threads and more overhead. |
| Examples include POSIX Pthreads, Marc C-threads and Solaris threads. | Examples include Windows 95/98/NT/2000, Solaris, Tru64 UNIX, BeOS and Linux. |

**Light-Weight Process**:

**Combined Approach**: Thread creation is done in the user space, while bulk of scheduling and synchronization of threads is done by the application. Example is Solaris.

**4.3.1 Many-to-One Model**

* *Description*: The many-to-one model maps many user-level threads to one kernel thread.
* *Efficiency*: Thread management is done by the thread library in user space, so it is efficient. It is totally portable and easy to do with few system dependencies.
* *Drawback*: The entire process will block if a thread makes a blocking system call – This is because only one thread can access the kernel at a time – multiple threads are unable to run on parallel on multi-core systems.
* *Example*: Very few systems currently use this model. Examples include Solaris Green Threads and GNU Portable Threads.

**4.3.2 One-to-One Model**

* *Description*: The one-to-one model maps each user thread to a kernel thread.
* *Efficiency*: It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call – This is because it allows multiple threads to run in parallel on multiprocessors.
* *Drawback*: Creating a user thread requires creating the corresponding kernel thread – the overhead of creating kernel threads can burden the performance of an application, therefore most implementations of this model restrict the number of threads supported by the system.
* *Example*: Windows, LinuxThreads and other systems where LWP creation is not too expensive.

**4.3.3 Many-to-Many Model**

* *Description*: The many-to-one model multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine. For example, an application may be allotted more kernel threads on a system with eight processing cores than on a system with eight processing cores than on a system with four cores.
* *Types*: In this model, the library has two kinds of threads: bound and unbound.
  1. *Bound Threads*: Bound threads are mapped each to a single lightweight process.
  2. *Unbound Threads*: Unbound threads may be mapped to the same LWP.
* *Efficiency*: The many-to-one model allows developers to create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.
* *Example*: It is used in the Solaris implementation of Pthreads and several other UNIX implementations. It also used in Windows with the *ThreadFibre* package. Otherwise, it is not very common.

**4.3.4 Two-Level Model**

* *Description*: It is a variation of the many-to-one model – it multiplexes many user-level threads to a smaller or equal number of kernel threads.
* *Difference*: It also allows a user-level thread to be bound to a kernel thread.
* *Example*: Solaris Operating System (versions older than Solaris 9).

**4.4 Thread Libraries**

**Thread Library**: A thread library provides the programmer with an API for creating and managing threads.

Implementation of a Thread Library:

|  |  |
| --- | --- |
| *User-Level Library* | *Kernel-Level Library* |
| The library is provided entirely in user space with no kernel support. | The library is implemented in a kernel-level library supported directly by the OS. |
| All code and data structures for the library exist in the user space. | Code and data structures for the library exist in the kernel space. |
| Invoking a function call in the library results in a local function call in user space and not a system call. | Invoking a function in the API for the library typically results in a system call to the kernel. |

Asynchronous ad Synchronous Threading:

|  |  |
| --- | --- |
| *Asynchronous Threading* | *Synchronous Threading* |
| Once the parent creates a child thread, the parent resumes its execution, so that the parent and child execute concurrently. | Once the parent creates one or more children, it must wait for all its children to terminate before it resumes – this is known as the **fork-join** strategy. |
| Each thread runs independently, and the parent thread need not know when its child terminates. | Once each thread has finished its work, it terminates and joins with its parent. |
| Because the threads are independent, there is typically little data sharing between threads. | Typically, it involves significant data sharing among threads. |

**4.4.1 Pthreads**

* Pthreads refer to the POSIX standard defining an API for thread creation and synchronization.
* This is a *specification* for thread behaviour, not an *implementation*. Operating-system designers may implement the specification in any way they wish.
* It is common in UNIX operating systems, like Linux, Mac OS X and Solaris.

**4.4.2 Windows Threads**

The technique for creating threads using the Windows Thread Library is similar to the Pthreads technique.

**4.4.3 Java Threads**

Threads are the fundamental mode of program execution in a Java program. All Java programs comprise of at least a single thread of control – even a simple Java program consisting of only a main() method runs as a single thread in the JVM.

There are two techniques for creating threads in a Java Program:

1. *Thread Class*: Create a new class that is derived from the Thread class and override its *run*() method.
2. *Runnable Interface*: Define a class that implements the runnable interface.

**4.5 Implicit Threading**

**Implicit Threading**: It is a strategy wherein the creation and management of threading is transferred from application developers to compilers and run-time libraries.

**4.5.1 Thread Pools**

Problems in Multithreading:

1. *Time*: The amount of time taken to create the thread is high.
2. *System Resources*: If all concurrent requests are allowed to be serviced in a new thread, no bound has been placed on the number of threads concurrently active in the system. Unlimited threads could exhaust system resources, such as CPU time and memory.

**Thread Pools**: A number of threads are created at process startup and placed into a pool, where they sit and wait for work. When a server receives a request, it awakens a thread from this pool.

1. *Thread is available*: The server passes it the request for service. Once the thread completes its service, it returns to the pool and awaits more work.
2. *Thread is not available*: The server waits until one becomes free.

Benefits of Thread Pools:

1. *Speed*: Servicing a request with an existing thread is faster than waiting to create a thread.
2. *Number of threads*: A thread pool limits the number of threads that exist at any one point.
3. *Separating the task to be performed from the mechanics of creating the task*: This allows us to use different strategies for running the task.

**4.5.2 OpenMP**

It is a set of compiler directives as well as an API for programs written in C, C++ or FORTAN that provides support for parallel programming in shared memory environments. OpenMP identifies **parallel regions** as blocks of code that may run in parallel.

**4.5.3 Grand Central Dispatch**

GCD is a technology for Apple’s Mac OS X and iOS operating systems. It is combination of extensions to the C Language, an API and a runtime library that allows application developers to identify sections of code to run in parallel.

**Blocks**: GCD identifies extensions to the C and C++ languages known as blocks. A block is a self-contained unit of work.

**Dispatch Queue**: GCD schedules blocks for run-time execution by placing them on a dispatch queue. When it removes a block from the queue, it assigns the block to an available thread from the thread pool it manages.

**Serial Dispatch Queue**: Blocks placed on a serial dispatch queue are removed in FIFO order. Once a block has been removed from the queue, it mist complete execution before another block is removed. Each process has its own serial queue known as its **main queue**.

**Concurrent Queue**: Blocks placed on a concurrent dispatch queue are also removed in FIFO order, but several blocks may be removed at a time, thus allowing multiple blocks to execute in parallel. According to priority, there are **low**, **default** and **high** concurrent dispatch queues. Priorities represent an approximation of the relative importance of the blocks.

**4.5.4 Other Approaches**

Several other parallel and concurrent libraries help in multithreading applications.

**4.6 Threading Issues**

**4.6.1 The fork() and exec() System calls**

* *Fork*: If one thread in a program calls fork(), the new process may either duplicate all threads or be single-threaded.
* *Exec*: If exec() is called immediately after forking, then duplicating all threads is unnecessary, as the program specified in the parameters to exec() will replace the process. In this instance, duplicating only the calling thread is appropriate. If, however, the separate process does not call exec() after forking, the separate process should duplicate all threads.

**4.2 Signal Handling**

**Signal**: A signal is used in UNIX systems to notify a process that a particular event has occurred. A signal may be received synchronously or asynchronously.

**Synchronous Signals**: They are delivered to the same process that performed the operation that caused the signal.

**Asynchronous Signals**: When a signal is generated by an event external to a running process, that process receives the signal asynchronously. An asynchronous signal is sent to another process.

Signal 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.

*Signal Handlers:*

1. **Default Signal Handler**: Every signal has a default signal handler that the kernel runs when handling that signal.
2. **User-Defined Signal Handler**: The default signal handler’s action can be overridden by a user-defined signal handler that is called to handle the signal.

*Where should a signal be delivered?*

1. Deliver the signal to the thread to which the signal applies.
2. Deliver the signal to every thread in the process.
3. Deliver the signal to certain threads in the process.
4. Assign a specific thread to receive all signals for the process.

**Asynchronous Procedure Calls (APCs)**: The APC facility enables a user thread to specify a function that is to be called when the user thread receives notification of a particular event. It is more straightforward than UNIX, since APC is delivered to a particular thread rather than a process. It is provided by *Windows*, which does not explicitly provide support for signals.

**4.6.3 Thread Cancellation**

**Thread Cancellation**: It involves terminating a thread before it has completed.

**Target Thread**: A thread that is to be cancelled is referred to as the target thread.

*Scenarios wherein cancellation of thread occurs:*

1. **Asynchronous cancellation**: One thread immediately terminates the target thread.
2. **Synchronous cancellation**: The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

*Difficulty in cancellation*: Resources have been allotted to a cancelled thread, or a thread is cancelled while in the midst of updating data it is sharing with other threads. This becomes especially troublesome with asynchronous cancellation. Often, the system will reclaim resources (but not all) from a cancelled thread.

**Deferred Cancellation**: One thread indicates that a target thread is to be cancelled, but cancellation occurs only after the target thread has checked a flag to determine whether or not it should be cancelled. The thread can perform this check at a point at which it can be cancelled safely – this is known as a **cancellation point**.

**Clean-up handler**: If a cancellation request is found to be pending, a function known as a clean-up handler is invoked. This function allows any resources a thread may have acquired to be released before the thread is terminated.

**4.6.4 Thread-Local Storage**

**Thread Local Storage**: Threads belonging to a process share the data of the process. However, in some circumstances, each thread might need its own copy of certain data. Such data is called thread-local storage. For example, in a transaction processing system, each transaction is assigned a unique identifier.

*Difference between Thread Local Storage and Local Variables*: Local variables are visible only during a single function invocation, whereas TLS data are visible across function invocations.

*TLS Data and Static Data*: TLS is similar to static data. The difference is that TLS data are unique to each thread.

*Examples*: Windows, P-Thread and Java provide Thread Local Storage.

**4.6.5 Scheduler Activations**

*Problem*: Communication between the kernel and the thread library is required by the many-to-many and two-level models. Such coordination allows the number of kernel threads to be dynamically adjusted to ensure best performance.

**Lightweight Process**: Systems implementing the many-to-one or the two-level model place an intermediate data structure between the user and kernel threads, known as lightweight process. Each LWP is attached to a kernel thread (but a kernel may not be bound to a LWP), and it is kernel threads that the operating system schedules to run on its physical processors.

**Scheduler Activation**: It is a scheme for communication between the user-thread library and the kernel. The kernel provides an application with a set of virtual processors (LWPs), and the application can schedule user threads onto an available virtual processor.

**Upcall**: The kernel must inform an application about certain events – this procedure is known as upcall, and is handled by the thread library with an **upcall handler**. One event that triggers an upcall occurs when an application thread is about to block. The upcall handler saves the state of the blocking thread and schedules another thread that is eligible to run on the new virtual processor.

**4.7 Operating System Examples**

**4.7.1 Windows Threads**

Windows uses the one-to-one mapping, where each user-level thread maps to an associated kernel thread.

Components of a Thread:

1. *Thread ID*: It uniquely identifies the thread.
2. *Register Set*: It represents the status of the processor.
3. *User Stack*: It is employed when the thread is running in user mode.
4. *Kernel Stack*: It is employed when the thread is running in kernel mode.
5. *Private Storage Area*: It is used by various runtime libraries and dynamic link libraries (DLLs).

**Context of thread**: The register set, stacks and private storage area are known as the context of the thread.

**4.7.2 Linux Threads**

**Task**: Linux does not distinguish between processes and threads. It uses the term task – while referring to flow of control within a program.

When *clone()* is invoked, it is passed a set of flags to determine how much sharing is to take place between the parent and child tasks. By default, if no flags are passed, no sharing takes place, resulting in functionality similar to that provided by the *fork()* system call.

S

S

s