THREADS AND MULTITHREADING

Objectives

Introduction to Threads

- A thread is a basic unit of CPU utilization.
- Threads form the foundation of multithreaded systems.
- Threads allow a single process to perform multiple tasks concurrently.

A web browser might have one thread for displaying content and another for downloading files.

Thread APIs

- Discussion of Pthreads, Windows, and Java thread libraries.
- These APIs provide mechanisms to create, manage, and synchronize threads.
- Each API has its own syntax and features, but the underlying concepts are similar.

Pthreads uses functions like `pthread_create()` and `pthread_join()`, while Java uses the `Thread` class.

Implicit Threading

- Exploration of strategies for implicit threading.
- Techniques to create and manage threads without explicit programmer intervention.
- Examples include thread pools, OpenMP, and Grand Central Dispatch.

Using a thread pool, the programmer simply submits tasks, and the pool manages the threads that execute them.

Multithreaded Programming Issues

- Examination of challenges in multithreaded programming.
- Issues such as race conditions, deadlocks, and thread safety.
- Requires careful synchronization and resource management.

Two threads attempting to increment the same counter simultaneously can lead to data corruption if not properly synchronized.

Operating System Support

- Covering OS support for threads in Windows and Linux.
- How the OS schedules and manages threads.
- Different scheduling policies and thread priorities.

Linux uses the `clone()` system call to create new threads, while Windows uses `CreateThread()`.

BENEFITS OF MULTITHREADING

Multithreading Benefits

Responsiveness

- Allows a program to continue running even if part of it is blocked.
- Crucial for user interfaces as it prevents the entire application from freezing.

Consider a web browser. One thread handles downloading a large file, while another thread continues to respond to user interactions like scrolling or clicking links.

Resource Sharing

- Threads share the resources (memory, files, etc.) of their parent process.
- Easier and more efficient than inter-process communication methods like shared memory or message passing.

Multiple threads in a word processor can access and modify the same document data without needing explicit data transfer mechanisms.

Economy

- Creating and managing threads is generally less expensive than creating and managing processes.
- Thread switching has lower overhead compared to process context switching.

Spawning a new process involves allocating a separate memory space and other resources, while creating a new thread uses the existing process's resources.

Scalability

- A multithreaded process can effectively utilize multiple processors in a multiprocessor architecture.
- Threads can run in parallel on different CPUs, increasing overall performance.

A video editing application can use multiple threads to simultaneously render different parts of a video on different CPU cores, drastically reducing rendering time.

MULTICORE PROGRAMMING

Multicore Programming Challenges

Dividing Activities

- Break down the problem into smaller, independent tasks.
- Distribute tasks across multiple cores for parallel execution.
- Goal is to maximize core utilization and reduce overall execution time.

Imagine sorting a large array. You could divide the array into smaller sub-arrays, sort each sub-array concurrently on different cores, and then merge the sorted sub-arrays.

Balance

- Ensure each core has an equal workload.
- Uneven workload distribution leads to some cores being idle while others are busy.
- Careful task assignment is crucial for optimal performance.

If one core is responsible for a much larger portion of data processing, it will become a bottleneck, limiting the overall speedup from parallelism.

Data Splitting

- Dividing the data to be processed across different cores.
- Data splitting needs to be efficient to minimize overhead.
- How the data is split can significantly affect performance.

In image processing, you could divide an image into regions and process each region on a separate core.

Data Dependency

- Identify and manage dependencies between tasks.
- If one task depends on the output of another, they cannot run simultaneously.
- Synchronization mechanisms are needed to coordinate access to shared data.

Task A calculates a value that Task B needs. Task B cannot start until Task A completes.

Testing and Debugging

- Multicore programs are inherently more complex to test and debug.
- Race conditions and deadlocks are common problems.
- Requires specialized tools and techniques.

Debugging tools for finding race conditions, where multiple threads try to access and modify shared data concurrently, leading to unpredictable results.

Parallelism vs. Concurrency

- Parallelism = true simultaneous execution of tasks.
- Concurrency = ability to manage multiple tasks, possibly interleaved.
- Multicore systems enable true parallelism.

A single-core processor can only execute one instruction at a time. It can, however, provide concurrency by quickly switching between tasks, giving the illusion of parallelism. A multicore processor can truly execute multiple instructions at the same time, offering true parallelism.

MULTICORE PROGRAMMING PARALLELISM

Multicore Programming (Cont.)

Data Parallelism

- Distributes subsets of the same data across multiple cores.
- Each core performs the same operation on its subset.
- Focuses on dividing the *data* workload.

Imagine calculating the average temperature of a large grid. Each core could calculate the average temperature of a section of the grid.

Task Parallelism

- Distributes threads across cores.
- Each thread performs a *unique* operation.
- Focuses on dividing the *tasks* to be performed.

One thread reads data from a file, another thread processes the data, and a third thread writes the processed data to another file, all concurrently.

Threads and Architectural Support

- Increasing thread count necessitates better architectural support.
- CPUs have cores *and* hardware threads.
- Hardware threads allow a single core to appear as multiple logical cores.

Oracle SPARC T4 with 8 cores, and 8 hardware threads per core is given as an example. This CPU would present 64 logical cores to the operating system.

MULTITHREADING MODELS

Multithreading Models

Many-to-One Model

- Many user-level threads are mapped to a single kernel thread.
- Thread management is done by the thread library in user space, so it is efficient.
- The entire process will block if a thread makes a blocking system call.
- Multiple threads are unable to run in parallel on multicore systems.

Green threads (early Java threads) and some user-level thread libraries use this model.

One-to-One Model

- Each user-level thread maps to a kernel thread.
- Provides more concurrency than the many-to-one model.
- When a thread makes a blocking system call, other threads can continue to run.
- Creating a user thread requires creating the corresponding kernel thread.
- The overhead of creating kernel threads can burden the performance of the system.
- Most implementations of this model place a limit on the number of threads.

Linux, Windows, and Solaris use the one-to-one model.

Many-to-Many Model

- Multiplexes many user-level threads to a smaller or equal number of kernel threads.

- Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.
- When a thread performs a blocking system call, the kernel can schedule another thread for execution.

The Solaris operating system used to implement this model.

MANY-TO-ONE THREADING MODEL

Many-to-One Threading Model

Mapping and Blocking

- Many user-level threads are mapped to a single kernel thread.
- If one thread blocks, the entire process blocks.
- True parallelism isn't possible on multicore systems because only one thread can access the kernel at a time.

Imagine a busy kitchen (a process) with many chefs (user-level threads) working on different dishes. But there's only one stove (the kernel thread). If one chef needs the stove and it's blocked (e.g., waiting for a pot to boil over), all the other chefs have to wait too.

Usage

- This model is not widely used in modern systems.

Older implementations of Solaris Green Threads and GNU Portable Threads used the many-to-one model, but they are less common now.

THREADING MODELS

One-to-One Model

Defining Characteristics

- Each user-level thread corresponds directly to one kernel thread.
- Creating a user-level thread inherently creates a corresponding kernel thread.
- Offers greater concurrency compared to the many-to-one model.
- The number of threads a process can have might be limited due to the overhead associated with creating and managing kernel threads.

Imagine a single-lane road where each car (user-level thread) has its dedicated lane (kernel thread). Each car can move independently, allowing for parallel traffic flow, but adding more cars also adds significantly to the road's construction cost and management.

Examples

- Windows operating system
- Linux operating system
- Solaris 9 and later

OPERATING SYSTEM CONCEPTS - MANY-TO-MANY MODEL

Many-to-Many Threading Model

Definition

- Many user-level threads are mapped to many kernel-level threads.
- Provides greater concurrency by mapping multiple user threads to a pool of kernel threads.
- The operating system can create a sufficient number of kernel threads, avoiding blocking when a user thread makes a blocking system call.

Solaris OS before version 9 and Windows with the ThreadFiber package implemented the many-to-many threading model.

Advantages

- Supports more concurrency than the many-to-one model.
- Allows a user thread to block without blocking the entire process.
- Multiple threads can run in parallel on multiprocessors.

Disadvantages

- Still requires some coordination between user and kernel threads.
- Can be complex to implement correctly.