**Chapter 25: Dialogue**

A **thread** is kind of like an independent agent running around in this program, doing things on the program’s behalf. But these threads access memory, and for them, each spot of memory is kind of like one of those peaches. If we don’t coordinate access to memory between threads, the program won’t work as expected.

OS must support multi-threaded applications with primitives such as locks and condition variables.

**Chapter 26: Concurrency: An Introduction**

A **multi-threaded** program has more than one point of execution (i.e., multiple PCs, each of which is being fetched and executed from). Threads **share** the same address space and thus can access the same data.

Each thread has its own private set of registers it uses for computation. Thus, if there are two threads that are running on a single processor, when switching from running one (T1) to running the other (T2), a **context switch** must take place. The context switch is similar to that between processes as T1 must be saved and T2 must be restored before running. We save the state to a **process control block (PCB)**. We will need one or more **thread control blocks (TCBs)** to store the state of each thread of a process.

Diagram

Description automatically generated

In the multi-threaded process, each thread runs independently. Instead of a single stack in the address space, there will be one per thread.

Any stack-allocated variables, parameters, return values, and other things that we put on the stack will be placed in what is sometimes called thread-local storage, i.e., the stack of the relevant thread.

**26.1 Why Use Threads?**

The first reason is **parallelism**. This makes the program runs faster because we divide the task among many threads.

The second reason is to avoid blocking program process due to slow I/O. Imagine when the program is performing some I/O task and we wish to do something else while waiting. Threading enables **overlap** of I/O with other activities within a single program, much like multiprogramming did for processes across programs.

We can use multiple processes instead of threads. However, threads share an address space and making it easy to share data. Processes are a more sound choice for logically separate tasks where little sharing of data structures in memory is needed.

**26.2 An Example: Thread Creation**

Text

Description automatically generated

After creating two threads, the main threads calls join, which waits for a particular thread to complete. The order that each thread gets to run depends on the **OS scheduler**.

**26.3 Why It Gets Worse: Shared Data**

The problem arises when two threads wish to update a global shared variable.

1. We wrap the thread creation and join routines to simply exit on failure; for a program as simple as this one, we want to at least notice an error occurred (if it did), but not do anything very smart about it.
2. Instead of using two separate function bodies for the worker threads, we just use a single piece of code, and pass the thread an argument (in this case, a string) so we can have each thread print a different letter before its messages.
3. The real problem is when two threads try to increase the same count 1 million time. The result will be different each time.

**26.4 The Heart Of The Problem: Uncontrolled Scheduling**

The problem is that the task to increase a variable by one is actually consists of three operations: read the data from an address, increase it by one and write it back to the address.

A scenario might happen that the threads read the same value at first and increment it an then try to update it with the wrong value.

The problem here is defined as **race condition**. Because multiple threads executing this code can result in a race condition, we call this code a **critical section.** To solve this problem, we use a **mutual exclusion**.

**26.5 The Wish For Atomicity**

Assume the instruction adds a value to a memory location, and the hardware guarantees that it executes **atomically**; when the instruction executed, it would perform the update as desired.

**Atomically**, in this context, means “as a unit”, which sometimes we take as “all or none.”

What we will instead do is ask the hardware for a few useful instructions upon which we can build a general set of what we call **synchronization primitives**. By using this hardware support, in combination with some help from the operating system, we will be able to build multi-threaded code that accesses critical sections in a synchronized and controlled manner, and thus reliably produces the correct result despite the challenging nature of concurrent execution.

**26.6 One More Problem: Waiting For Another**

There is another common interaction that arises, where one thread must wait for another to complete some action before it continues. This interaction arises, for example, when a process performs a disk I/O and is put to sleep; when the I/O completes, the process needs to be roused from its slumber so it can continue.