Introduction

So far, we've observed the evolution of the OS's essential abstractions. A single hardware CPU can be turned into multiple virtual CPUs, giving the illusion of multiple programs running simultaneously. We've also shown how to give each process the illusion of a huge, private virtual memory while the OS is surreptitiously multiplexing address spaces across physical memory (and sometimes, disk).

Threads are a novel abstraction for a single operating process. Instead of a single execution point, multi-threaded programs have several execution points (for example, multiple PCs) (e.g., a single machine where instructions are fetched from and run). Each thread is very much like a separate process, except that they share the same address space and so can access the same data.

A thread's state is thus quite similar to a process's. An instruction fetches a program counter (PC). Because each thread has its own set of computation registers, moving between running one (T1) and the other (T2) requires a context switch (T2). This is because we must save T1's register state and restore T2's register state before starting T2. Previously, state was saved to a process control block (PCB); now, each thread in a process will need its own thread control block (TCB). The address space remains unchanged when we move context between threads and processes (i.e., there is no need to switch which page table we are using).

The stack is another major distinction between threads and processes. This depiction of the address space of a classic (now single-threaded) process has a single stack at the bottom of the address space (Figure 26.1, left).

For each task, a multi-threaded process may call into numerous functions. So instead of one stack per thread, there will be two. Consider a two-threaded process; the resulting address space appears different (Figure 26.1, right).

In this image, two stacks are seen stretched across the process's address space. So, any stack-allocated variables, parameters, return values, and other items are stored in thread-local storage, i.e., the thread's stack.

Notice how this disrupts our lovely address space layout. Before the stack and heap could grow independently, problems occurred when the address space ran out. This is no longer a nice condition. Stacks don't have to be huge, so this is usually fine (the exception being in programs that make heavy use of recursion).

Why Use Threads?

Before diving into the nitty-gritty of threads and multi-threaded programming, let’s address a more basic question. Why utilize threads at all?

Threads are useful for at least two reasons. First, parallelism. Imagine building a program that adds two huge arrays together or increases the value of each element in the array by a certain amount. On a single CPU, the task is simple: perform each operation. Assume you’re running the program on a multi-processor system. If so, you may greatly speed up the process by having each processor do a piece of the job. Parallelizing a single-threaded application across many CPUs is a standard approach to make current programs perform faster.

The second reason is more subtle: to avoid slow I/O obstructing program progress. Imagine writing a program that waits for I/O to complete: sending or receiving messages, explicit disk I/O, or even (implicitly) a page fault. Instead of waiting, your software may want to use the CPU to compute or send more I/O requests. While one thread in your program waits for I/O, the CPU scheduler can swap to other threads ready to start and perform something productive. Threading also allows I/O to be shared with other activities within a program, similar to how multiprogramming shared processes between programs; thus, many modern server-based applications (web servers, database management systems, etc.) use threads.

In all circumstances, you could use several processes instead of threads. Threads share an address space and so make data sharing straightforward, making them a perfect fit for these programs. Processes are better for logically independent tasks that require less memory sharing.

An Example: Thread Creation

Let’s dig into the details. Say we wanted to execute a program that spawns two threads, each of which prints “A” or “B.” Here’s the code.

To run mythread(), the main program starts two threads (A or B). The scheduler can start a thread immediately (or it can be put in the “ready” state but cannot run yet). On a multiprocessor, the threads could run simultaneously, but let’s not worry about that yet.

After constructing T1 and T2, the main thread executes pthread join(), which waits for a thread to finish. The main thread then writes “main: end” and exits after T1 and T2 have finished running. So, this run used three threads: main, T1 and T2.

Examining the program’s possible execution order. The execution diagram (Figure 26.3) shows time increasing downward, with each column representing a separate thread (the main, Thread 1, or Thread 2).

This is not the only possible ordering. Depending on whatever thread the scheduler decides to run at the time, a particular set of instructions has many alternatives. If a thread is formed, it may run immediately, as seen in Figure 26.4.

Also, if the scheduler schedules Thread 2 first, even though Thread 1 was previously formed, we can see “B” written before “A.” This is shown in Figure 26.5, where Thread 2 gets to shine before Thread 1.

The system establishes a separate thread of execution for the routine that is being called, and it runs independently of the caller, possibly before returning from thread creation, but possibly much later. A viable mechanism is presumably implemented by the OS scheduler, but knowing what will execute next is difficult.

As this example shows, threads complicate life: it’s hard to predict what will execute when! Computers are complex enough without concurrency. But, sadly, concurrency makes things worse. Worse.

Why It Gets Worse: Shared Data

The simple thread example above explained how threads are created and how the scheduler selects how to operate them. It doesn’t demonstrate how threads interact while accessing shared data.

Imagine two threads updating a global shared variable. The code is in Figure 26.6.

A few notes on the code. In the first place, we wrap the thread creation and join algorithms to quit on failure, (e.g., just exit). As a result, Pthreadcreate() just calls pthreadcreate() and checks the return code; if not, it produces a message and exits.

Second, we use a single function body for the worker threads and feed them an input (in this case, a string) such that each thread prints a distinct letter before its message.

Unfortunately, even on a single processor, this code doesn’t always get the desired outcome. We get:

Let’s do it again to see whether we’ve gone insane.

Each run is incorrect and produces a different result! So, why does this happen?

The Heart Of The Problem: Uncontrolled Scheduling

To understand why this happens, we must first understand the compiler's code sequence for updating counter. So we want to increase the counter by 1. So, on x86, the code may look like this:

This example assumes the variable counter is 0x8049a1c. The x86 mov instruction gets the memory value at the address and puts it into register eax first. It is then added to the eax register's contents (1 (0x1), and the eax register's contents are returned to memory at the same address.

Assume one of our two threads (Thread 1) accesses this code section and is about to add one to the counter. It first loads the counter's value (let's say 50) into its register eax. Thread 1: eax=50 So eax=51. Unfortunately, a timer interrupt occurs, and the OS saves the thread's state (PC, registers including eax, etc.) to its TCB.

Worse, thread 2 is chosen to run, inputting the same code. For the first instruction, it gets the counter value and stores it in its EAX (remember: each thread when running has its private registers; the registers are virtualized by the context-switch code that saves and restores them). The counter is still at 50, therefore Thread 2 has eax=50. Consider Thread 2 incrementing eax by 1 (eax=51) and then saving its contents into the counter (address 0x8049a1c). So the global variable counter currently has 51.

Thread 1 resumes running after another context switch. It had just done mov and add and is ready to do the last mov. Remember that eax=51. To save it to memory, the last mov instruction runs, setting the counter back to 51.

Simply put, the code to increment counter was executed twice, but counter, which started at 50, is now only 51. If this program was "correct," the variable counter should have been 52.

Let's look at an execution trace to better understand the issue. The preceding code loads at address 100 in memory, as seen below (notice for those used to lovely, RISC-like instruction sets: The mov operation uses 5 bytes of memory, but the add uses only 3 bytes.

Figure 26.7 shows the result of these assumptions (page 10). Assume the counter starts at 50 and follow along to ensure you understand.

This is a race condition (or, more particularly, a data race) where the outcome is dependent on the code execution timing. We get a negative result by chance (i.e., context switches that occur at un- timely points in the execution). Moreover, we may receive various results each time; so, instead of a clean deterministic calculation (like computers), we label this outcome indeterminate. It is likely to vary between runs.

This code is a key portion because several threads running it can produce a race scenario. A critical section is a block of code that accesses a common variable (or resource) and cannot be executed by multiple threads.

We want mutual exclusion for this code. If one thread operates within the critical part, the others are stopped.

The Wish For Atomicity

One solution would be to have more powerful instructions that handled everything we wanted in one step, eliminating the danger of a premature interrupt. What suppose we had a fantastic instruction like this:

Assume this instruction updates a memory address and that the hardware guarantees atomic execution. It couldn't be interrupted mid-instruction since the hardware guarantees that an interrupt means either the instruction hasn't run or it has run to completion. Isn't hardware lovely? As a unit, it signifies "all or none" in this context. So, we want to execute the three instructions atomically.

As stated previously, a single instruction would suffice. In general, we won't have such a directive. Imagine updating a concurrent B-tree; would we want hardware to allow a "atomic update of B-tree" instruction? No, not in a sensible instruction set.

To develop a large set of synchronization primitives, we shall instead ask the hardware for valuable instructions. Using this hardware support and the operating system's help, we may develop multi-threaded code that synchronizes and restricts access to important areas to dependably provide the proper output.

One More Problem: Waiting For Another

So far, this chapter has focused on one sort of inter-thread communication, accessing shared variables, and the necessity to provide atomicity for key areas. A common interaction occurs when one thread has wait for another to perform an action before proceeding. This interaction occurs when a process does a disk I/O and is put to sleep; when the I/O is finished, the process must be woken up to continue.

So, in the following sections, we'll look at how to enable atomicity and techniques for sleeping/waking in multi-threaded applications. It's fine if this doesn't make sense now! When you read the section on condition variables, it will. It is less OK if it does not arrive by then, and you should read that chapter again (and again) until it does.

Summary

Before we end, you may wonder why we are studying this in OS class. The OS was the first concurrent application, and numerous techniques were developed for use within the OS. With multi-threaded processes, application programmers have to think about this too.

Consider the scenario of two processes. Assume they both call write() to write data to the file (i.e., add the data to the end of the file, thus increasing its length). To do so, both must allocate a new block, register it in the file's inode, and resize the file (we'll learn more about files in the third part of the book). Were it not for the fact that an interrupt can occur at any time, the code updating shared structures (e.g., a bitmap for allocation or the file's inode) would be important. An untimely interruption creates the issues listed above. To use the proper synchronization primitives, page tables, process lists, file system structures, and almost every kernel data structure must be carefully accessed.

These four concepts are so vital to concurrent code that we decided they deserved special mention.

- A critical section is a section of code that accessed a shared resource, usually a variable or data structure.
- A race condition (or data race) occurs when multiple threads attempt to update the same data structure at roughly the same time, resulting in unexpected (and perhaps undesirable) results.
- An indeterminate program contains one or more race conditions; its output varies from run to run depending on which threads were running. The outcome is thus not deterministic, something we usually expect from computer systems.
- Threads should use some kind of mutual exclusion primitives so that they do not race, and the program output is deterministic.