User-Mode Thread Implementation

Introduction

For a long time, processes were the only unit of parallel computation. There are two problems with this:

- processes are very expensive to create and dispatch, due to the fact that each has its own virtual address space and ownership of numerous system resources.
- each process operates in its own address space, and cannot share in-memory resources with parallel processes (this was before it was possible to map shared segments into a process' address space).

The answer to these needs was to create threads. A thread ...

- is an independently schedulable unit of execution.
- runs within the address space of a process.
- has access to all of the system resources owned by that process.
- has its own general registers.
- has its own stack (within the owning process' address space).

Threads were added to Unix/Linux as an after-thought. In Unix, a process could be scheduled for execution, and could create threads for additional parallelism. Windows NT is a newer operating system, and it was designed with threads from the start ... and so the abstractions are cleaner:

- a process is a container for an address space and resources.
- a thread is **the** unit of scheduled execution.

A Simple Threads Library

When threads were first added to Linux they were entirely implemented in a user-mode library, with no assistance from the operating system. This is not merely historical trivia, but interesting as an examination of what kinds of problems can and cannot be solved without operating system assistance.

The basic model is:

- Each time a new thread was created:
  - we allocate memory for a (fixed size) thread-private stack from the heap.
  - we create a new thread descriptor that contains identification information, scheduling information, and a pointer to the stack.
  - we add the new thread to a ready queue.
- When a thread calls *yield()* or *sleep()* we save its general registers (on its own stack), and then select the next thread on the ready queue.
- To dispatch a new thread, we simply restore its saved registers (including the stack pointer), and return from the call that caused it to *yield*.
- If a thread called *sleep()* we would remove it from the ready queue. When it was re-awakened, we would put it back onto the ready queue.
- When a thread exited, we would free its stack and thread descriptor.

Eventually people wanted preemptive scheduling to ensure good interactive response and prevent buggy threads from tieing up the application:

- Linux processes can schedule the delivery of (SIGALARM) timer signals and register a handler for them.
- Before dispatching a thread, the we can schedule a SIGALARM that will interrupt the thread if it runs too long.
- If a thread runs too long, the SIGALARM handler can *yield* on behalf of that thread, saving its state, moving on to the next thread in the ready queue.

But the addition of preemptive scheduling created new problems for critical sections that required before-or-after, all-or-none serialization. Fortunately Linux processes can temporarily block signals (much as it is possible to temporarily disable an interrupt) via the *sigprocmask(2)* system call.

Kernel implemented threads

There are two fundamental problems with implementing threads in a user mode library:

- what happens when a system call blocks

  If a user-mode thread issues a system call that blocks (e.g. *open* or *read*), the process is blocked until that operation completes. This means that when a thread blocks, all threads (within that process) stop executing. Since the threads were implemented in user-mode, the operating system has no knowledge of them, and cannot know that other threads (in that process) might still be runnable.

- exploiting multi-processors

  If the CPU has multiple execution cores, the operating system can schedule processes on each to run in parallel. But if the operating system is not aware that a process is comprised of multiple threads, those threads cannot execute in parallel on the available cores.

Both of these problems are solved if threads are implemented by the operating system rather than by a user-mode thread library.

Performance Implications

If non-preemptive scheduling can be used, user-mode threads operating in with a *sleep/yield* model are much more efficient than doing context switches through the operating system. There are, today, *light weight thread* implementations to reap these benefits.

If preemptive scheduling is to be used, the costs of setting alarms and servicing the signals may well be greater than the cost of simply allowing the operating system to do the scheduling.

If the threads can run in parallel on a multi-processor, the added throughput resulting from true parallel execution may be far greater than the efficiency losses associated with more expensive context switches through the operating system. Also, the operating system knows which threads are part of the same process, and may be able to schedule them to maximize cache-line sharing.

Like preemptive scheduling, the signal disabling and reenabling for a user-mode mutex or condition variable implementation may be more expensive than simply using the kernel-mode implementations. But it may be possible to get the best of both worlds with a user-mode implementation that uses an atomic instruction to attempt to get a lock, but calls the operating system if that allocation fails (somewhat like the *futex(7)* approach).