Thread

Contents

Introduction

In operating systems, threads are the basic unit of execution and concurrency of various tasks pending in the system. They are generally internal components of processes and share the same set of resources, i.e. file handles, shared-memory handles, POSIX signals, message-passing buffers and many more. But they can have specific sets of special resources in architectures like Windows NT, where the kernel can destroy *some resources corresponding to a thread when it terminates*.

Threads can see each other executing in a parallel fashion as they get selected by the scheduler, and then execute for a specific period of time. That means if the threads were designed to repeatedly print a specific number unique to each thread, you would see them being printed in a random fashion. But threads can concurrently execute in real-time on multi-processor systems only, because multiple CPUs are executing the tasks simultaneously as opposed to uni-processor systems where one CPU executes each task one-by-one.

Advantages

All modern operating systems implement threads with deep support. They export kernel calls for configuring how the kernel manages various threads. Threads have many advantages over traditional processes and are used in newer applications:

1. **Concurrency** - Threads allow multiple tasks to execute in a parallel fashion and speed up the various work of the program. For example, two worker threads can be parsing two different CSS stylesheets in a browser while loading a page. This increases the output and responsiveness of the program.

2. **Better Resource Usage** - When work is divided into separate processes, a huge overhead is incurred as compared to dividing the work between threads that live in the same address space under one process. It also reduces context switches and can provide better cache usage for the program.

3. **User-level Management of Threads** - Threads can also be managed at a user-level which eliminates the need for a context switch to the kernel. This greatly improves performance with the the help of Scheduler Activations (see Scheduling) and the M:N threading model (which we will talk about later in this article).

4. **Simplicity** - Creating a single program which effectively utilizes a thread API, is very simple compared with creating separate work-processes and corresponding programs, using inter-process communication to inter-link them.

Complexity of Threads

Multithreading in programs and the operating system itself can lead to the following undesirable conditions:

1. **Race Conditions & Synchronization Overhead** - When many threads are working on the same data structure in memory, then they must somehow serialize their changes with synchronization techniques. This greatly impacts performance when the number of competing threads is huge, especially when concurrency bugs cause deadlocks, livelocks or other race conditions. For example, if threads 'A' and 'B' require both the resources 'R1' and 'R2', and 'A' locks 'R1' first and 'B' locks 'R2' first. Now both are competing for the other resource, both of which are already locked. This results into a complete stoppage of execution and is called a deadlock.

2. **Reliability** - Multithreading is indeed complex compared to the single-threaded model. Also, if one thread throws an exception (not C++) which the system cannot handle then the whole process may be terminated.

Kernel Threads

The task scheduler in the kernel handles kernel-level threads. These threads are held in a data structure by the kernel and are managed by the kernel. They can be blocked for I/O by the kernel and easily destroyed from external processes.

Upon entry into the kernel, the kernel stack for the thread is loaded and the user stack, along with its execution state, is saved. Each thread may have its own stack or share a set of stacks.

Multiple Kernel Stacks

The main advantage of each thread having its own kernel stack is that system calls can block inside of the kernel and easily resume later from that point. If a page fault or interrupt were to occur during a system call, then it's possible to switch to another context and back, and later complete the system call. However, a large number of threads could tie up a significant amount of memory, most of which would sit unused at any given time. Further, the constant switching of kernel threads may lead to higher cache misses and thus poorer performance.

Single Kernel Stack

An alternative is to have one kernel stack per processor and let threads share the stack(s). This greatly reduces the amount of memory that must be allocated. The thread gets control of its processor's kernel stack when entering the kernel and relinquishes it upon a thread switch. Since the stack pointer is reset to the top, the stack is effectively destroyed and created anew. However, this means that threads cannot easily block or be preempted inside of the kernel. For example, a microkernel may need to send a message to a server that handles page faults, but upon the switch to the server thread, the stack of the faulting thread would be wiped out. The kernel must either provide a mechanism to restart interrupted system calls (using continuations, for example) or guarantee that system calls will not block, be preempted, or fault (any such condition would be fatal.)

Variable Kernel Stacks

A compromise between multiple stacks and a single stack is to use a single stack per processor for all threads, but allocate a new stack when a thread blocks or is preempted inside of the kernel. The blocked thread takes ownership of the old stack, and the other threads share the new one. When the thread is unblocked and completes its operation inside of the kernel, its stack is deallocated. This allows blocking inside of the kernel while minimizing memory usage. The main issue with this approach is that memory for stacks has to be allocated and deallocated every time threads are switched, possibly at a time when the kernel must handle an interrupt in a timely manner.

User Threads

The user-level threads are held by the user-level scheduler and are not seen by the kernel. They can be created and destroyed in userspace and thus are of even lower overhead than kernel threads.

Threading Models

Threads can be implemented in a system using these three methods:

1:1 Model

Each that the process uses in user-space could be scheduled in the kernel. This means that the scheduler will actually hold structures about all threads that the process is using.

Advantages

When all user-level threads are backed by unique kernel threads, then the program gets the following benefits -

1. All threads can execute concurrently on separate processors. The scheduler can assign each thread to a processor and if possible all of them can really execute in a parallel fashion.

2. Depending on the kernel to manage threads is simple compared to managing user-level threads separately.

3. Pausing for some specific time is easier with kernel threads because the kernel can easily access the APIC timer.

4. The scheduler can be built with a simplistic model without the concept of the user-level scheduler.

Disadvantages

When all user-level threads are backed by kernel threads, then the their creation, destruction, pausing, waiting, blocking, etc. become expensive operation due to a context switch to the kernel. It also causes a load-imbalance at the process level. A process with a huge amount of threads running could harm the system as it will take up most of the vital time of other important tasks. This problem can be dealt with **group scheduling.**

1:n Model

In this model, all the user-level threads are directly mapped to one kernel thread. For the kernel, this process will look like a single-threaded program. But in user-mode the kernel-thread will directly execute the user-level scheduler which in turn selects a thread.

Advantages

This method reduces the overhead of creating, destroying, blocking, etc. user-level threads as all threads can be managed in user-space without any context switch to the kernel. It also allows the process to have a custom scheduler for its tasks and could also help in co-operative scheduling

Disadvantages

Threads cannot easily be blocked for external resources without stopping other threads. As the kernel can only pause a kernel thread, the blocking of a user-level thread will in turn take down the process's output if it takes too long.

M:N Model

This is a theoretically ideal model for threading. There are 'M' number of threads in user-space backed by 'N' number of kernel threads, provided M > N. It was previously implemented on FreeBSD but was deprecated due to its complexity.

Advantages

This method allows fast user-level thread management with the functionality of kernel-threads. Also, it reduces the resource usage of thread structures in kernel. For example, if there are two CPUs in the system and a process uses 8 threads. It would be more practical to map groups of 4 threads to 2 kernel threads. The priority of these two threads could be increased as the mapping

increases, and that will allow a dynamic relationship between user-level and kernel threads. If more than two-kernel threads are used for running threads, then they will still not be able to execute concurrently due to the limitation of two CPUs.

Disadvantages

The M:N model is quite complicated. It involves co-operation from both the user-level side and the kernel side. In practical tests, this complexity has many times failed for general applications to give substantial benefit over the 1:1 model. This is why it has been deprecated in systems like Solaris and FreeBSD.

See Also

- Warton, Matthew. *Single Kernel Stack L4*. 2 Nov 2005. https://sjmulder.nl/dl/pdf/unsorted/2005%20-%20Warton%20-%20Single%20Kernel%20Stack%20L4.pdf
- Understanding Threads (https://sites.ualberta.ca/dept/chemeng/AIX-43/share/man/info/C/a_doc_lib/aixprggd/genprogc/understanding_threads.htm)

Retrieved from "https://osdev.wiki/wiki/Thread"

Content is available under CC0 Public Domain unless otherwise noted.