Multithreading

An Operating System Analysis

Author: Kevin Haghighat

Date: December 1st 2008

1. OVERVIEW

Threads are an inherit part of software products as a fundamental unit of CPU utilization as a basic building block of multithreaded systems. The use of threads has evolved over the years from each program consisting of a single thread as the path of execution of it. The notion of multithreading is the expansion of the original application thread to multiple threads running in parallel handling multiple events and performing multiple tasks concurrently. Today's modern operating systems foster the ability of multiple threads controlled by a single process all within the same address space.

Multithreading brings a higher level of responsiveness to the user as a thread can run while other threads are on hold awaiting instructions. As all threads are contained within a parent process, they share the resourses and memory allocated to the process working within the same address space making it less costly to generate multiple threads vs. Processes. These benefits increase even further when executed on a multiprocessor architechture as multiple threads can run in parallel across multiple processors as only one process may execute on one processor.

Threads divide into two types: **user-level threads** – visible to developers but unknown to the kernel, and **kernel-level threads** – managed by the operating system's kernel. Three models identify the relationships between user-level and kernel-level threads: **one-by-one, many-to-one,** and **many-to-many**.

This report explores various notions related to systems with multithreading capability, including POSIX, Win32, and Java thread libraries.

Challenges associated to multithreaded program development explored in this report are those of thread cancellation, singnal handling, thread-specific data, and symantics of necessary system calls.

TABLE OF CONTENTS

2. INTRODUCTION

Extensively simplified, a thread is the path that a process or application takes during its execution. Today's operating systems facilitate a multithread environment. This report will explore and discuss various concepts and issues associated with threads and multithreaded operating systems. It will discuss the root three models, followed by a few thread libraries, explore potential issues, application programming interfaces (APIs), followed by some examples of multithreaded operating systems and their thread support mechanism.

Motivation

Traditionally, programs are single-path execution, hence a single thread. This practice would have made the production of today's software production impossible as the need of speed required programs to perform multiple tasks and events at the same time. With traditional turn-by-turn game, such as tic-tac-toe or chess, the traditional approach works fine, however with new age multitasking programs where multiple events need to run in parallel, the traditional approach proves useless.

Benefits

Responsiveness: Multithreading allows a process to keep running even if some threads within the process are stalled, working on a lengthy task, or awaiting user interaction. Using a digital alarm clock as an example of a process, the thread of keeping track of time, continues while an alarm is sounding while another awaits it's time to activate.

Cost Effective: Memory and resource allocation to process creation remains costly where as threads share the resources allocated to the process they reside in making it less costly to make threads or move them from on process to another.

Resource Distribution: The inherit property of sharing memory and resources of the parent process fosters the ability of having multiple treads occupying the same address space.

Cross-Processor Distribution: The benefits of multithreading are multiplied as the number of available processors increase opposite to single threading where only one processor is used. In a multiprocessor architecture, running of threads can distribute across multiple processors in parallel thereby increasing efficiency.

3. MODELS

Threads divide into two types, user threads and kernel threads. User threads are user-level threads handled independent from and above the kernel and thereby managed without any kernel support. On the other hand, the operating system directly manages the kernel threads. Nevertheless, there must be a form of relationship between user-level and kernel-level threads. There exist three established multithreading models classifying these relationships as **One-to-One, Many-to-One,** and **Many-to-Many**.

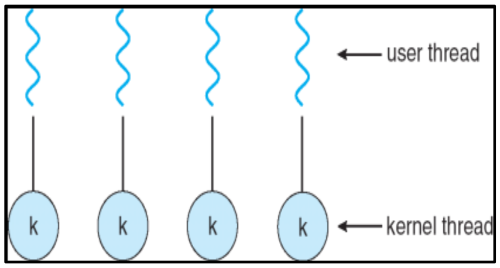

Figure 3.1: One-to-One Model

The one-to-one model (Figure 3.1) associates a single user-level thread to a single kernel-level thread. This type of relationship facilitates the running of multiple threads in parallel. However, this benefit comes with its own drawback such that generation of every new user thread must include the creation of a corresponding kernel thread causing an overhead, which can hinder the performance of the parent process. Windows series and Linux operating systems try to tackle this problem by limiting the growth of the thread count.

The many-to-one model (Figure 3.2) associates all user-level threads to a single kernel-level thread. This type of relationship facilitates an effective context-switching environment, easily implementable even on simple kernels with no thread support. The down side is that since there is only one kernel-level thread scheduled at any given time, this model cannot take advantage of the hardware acceleration offered by multi-threaded processors or multi-processor systems.

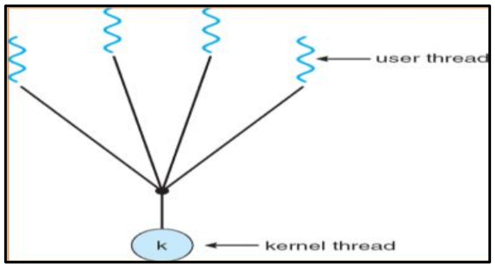

Figure 3.2: Many-to-One Model

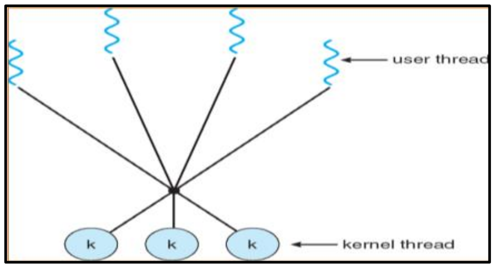

Figure 3.3: Many-to-Many Model

The many-to-many model (Figure 3.3) is a compromise between the last two models. In this model, a number of user-level threads are associated to an equal or smaller number of kernel-level threads. The requirement of changing code in both kernel and user spaces presents a level of complexity not present in the previous models. Similar to the many-to-one model, this model presents an effective context-switching environment as it keep away from system calls. The heightened complexity presents the potential for priority inversion and suboptimal scheduling with little coordination between the user and kernel schedulers.

4. LIBRARIES

Thread libraries provide the means of thread generations and management as APIs in the development of multithreaded applications. The implimentation of such libraries are performed via two principal approaches: user and kernel levels.

In the user level approach, all the actual code and data structures are held within the user space such that invoking a library method will result in a call to a local method within the user space rather than a system call.

In the kernel level approach, the actual code and data structures are held within the kernel space such that invoking a library method results in a system call to the kernel since kernel level libraries are implemented by the operating system itself.

The main thread libraries used today are POSIX Pthreads, Win32 threads and Java threads.

Pthreads – POSIX Pthreads are commonly used in Linux systems. Programmers create and manage Pthreads in both user and kernel levels.

Win32 Threads – Implemented by Windows NT operating system and above, they foster high-performance multhithreading. This type of thread belongs to the kernel level.

Java Threads – In Java, thread creation and management is done within the program in 2 forms:

1) Extending a class where a child class inherits methods and variables from a single parent class as the most common form of creating Java threads.

2) Interfaces allow programmers to create an abstraction of future implementation of classes. This abstraction sets the stage while implemented interface classes perform the actual tasks while following the same sets of rules enforced by the interface.

5. CHALLENGES

Just as with any new programming intervention, when using the multithreading consept, one must keep in mind the pottential chanllenges it presents. Some of such challenges are outlined in this section.

System Calls – One of the issues to keep in mind is how a **system call** deals with threads contained in a process that is getting duplicated. Do the threads also get duplicated or does the duplicated process only posses a single thread? Some Unix systems provide the means of both methods of duplication.

Cancellations – There are times when we may want to terminate a thread before it completes its purpose. This type of termination is referred to as **thread cancellation**. Imagine a chess game, where multiple moves are evaluated via different threads in order to find the shortest path of victory based on possible future moves. In such scenario, since all threads are running concurrently, as soon as one thread has found a path of victory, the rest of the threads can be cancelled since the found path would be the shortest path to a checkmate. When cancelling a “target” thread, we can take on of two approaches. One is **asynchronous** cancellation, where one thread terminates another that could lead to orphan resources since the target thread did not have a chance to free them. And the other, **deferred cancellation**, where each thread keeps checking if it should terminate and if so, do so in an orderly fashion freeing system resources used by the terminating thread.

Signal Handling – Unix systems use **signals** to keep track of events which must follow the same path of excecution as depicted in Figure 5.1, regardless of their type being synchronous or asynchronous.

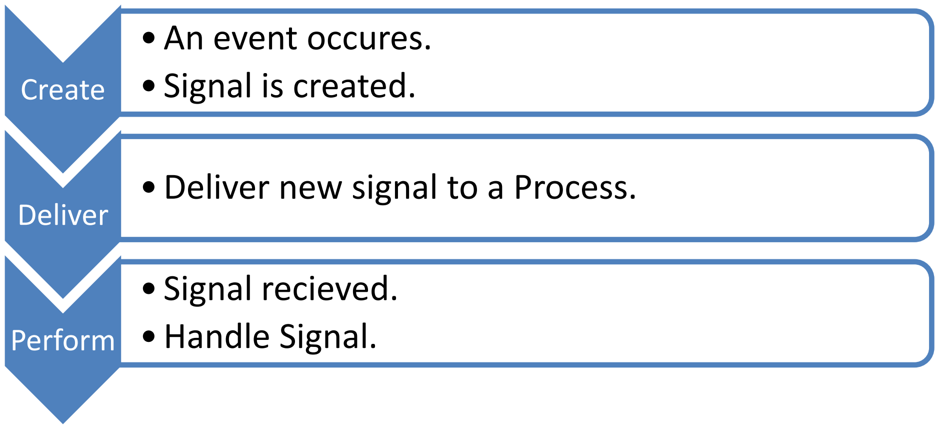

Figure 5.1: Path of an event execution.

An “illegal memory access” or “devide by 0” actions produce synchronous signals sent to the causing operation’s process. Asynchronous signals are those recieved as the result of an external event such as a keyboard commands like <ALT><F4> terminating a process, which are typically sent to another process.

Thread Pools – Eventhough creation of threads is more conservative than creating processes, unlimited threads can use up all the resources of a system. One way to keep this problem in check is the user of **thread pools**. The idea is to have a bunch of threads made upon the start of a process and hold them in a “pool”, where they await task assignment. Once a request is recieved, it is passed on to an available thread in the pool. Upon completion of the task, the thread then returns to the pool awaiting its next task. If the pool is empty, the system holds the requests until an available thread returned to the pool. This method limits the number of threads in a system to a managable size, most beneficial when the system does not posses enough resources to handle a high number of threads. In return, the performance of the system increases as thread creation is ofthen slower than reuse of an existing one.

Thread-Specific Data – The sharing of resources of the parent process does benefit multithreading programs, but in cases where a thread may need to hold it’s on copy of some data, called **thread-specified data**, it could be a downfall as well. The 3 main thread libraries discussed in this paper do provide support for such thread-specific handling which are often used as unique identifiers in transaction processing systems.

6. CONCLUSION

With multithreading having become an integral aspect of computing today, developers can produce multithreaded programs offering a wide variety of features and functionalities in a high performance environment while not over utilising system resources. Developers need to pay close attention to the potential challenges associated to the use of threads as they could greatly hinder their development progress.

I feel that Java threads merit special attention as Sun's Java development's cross platform methodology has heightened its place amongst the top programming languages.

7. ACKNOWLEDGEMENT:

I would like to extend my thanks to Academic Institutes whom offered access to publications on this topic. My wife for her support and understanding throughout the masters program. I would also like to offer my thanks for all researchers without their research and publications this paper would not have been possible. We would also like to extend my appreciation to Dr. Mahmoud Abaza for his guidance in writing this paper.

8. BIBLIOGRAPHY

- Silberschatz, A., Galvin, P. B., & Gagne, G. (2004). "**OPERATING SYSTEMS CONCEPTS, (7TH EDITION READING ED.).**" Canada: John Wiley & Sons Canada, Ltd.

- Arora, N. S., Blumofe, R. D., & Plaxton, G. C. (1998). "**THREAD SCHEDULING FOR MULTIPROGRAMMED MULTIPROCESSORS.**" In *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 119-129, New York, NY, USA. ACM Press.

- Drepper, U., & Molnar, I. (2003). "**THE NATIVE POSIX THREAD LIBRARY FOR LINUX.**" RedHat.

- Hilderink G.H., J.F. Broenink, & A.W.P. Bakkers. (1998). "**A NEW JAVA THREAD MODEL FOR CONCURRENT PROGRAMMING OF REAL-TIME SYSTEMS**", Real-time magazine, 98-1, pages 30-34

- Solomon D.A., & Russinovich M., (2000). "**INSIDE MICROSOFT WINDOWS 2000**", Microsoft Press, Redmond, WA

- Lee, E. A. (2006). "**THE PROBLEM WITH THREADS.**" IEEE., Computer, 39(5):33-42.

Thornley, J., Chandy, K. M., and Ishii, H. (1998). "**A SYSTEM FOR STRUCTURED HIGH-PERFORMANCE MULTITHREADED PROGRAMMING IN WINDOWS NT.**" In *Proceedings of the 2nd Conference on USENIX Windows NT Symposium - Volume 2* (Seattle, Washington, August 03 - 04, 1998). USENIX Association, Berkeley, CA, 8-8.