Thread models Semantics: Solaris and Linux M:N to 1:1 thread model

Ahmad Mohsin[1], Syed Irfan Raza[2], Syda Fatima[3]

[1, 2, 3] *Department of Computer Science and Engineering, Air University Multan Pakistan*
{ahmadspm, irfanrazanaqvi9, silenteyes36 }@gmail.com

Abstract. Multithreading has got significance with increasing demand for performance, speed and efficiency. As the hardware resources are becoming more powerful in terms of performance there is immense pressure on system software development side to optimize the hardware. Effective use of multithreading needs some mechanism to implement. Thread models have greater impact on application execution time, and I/O operations. Work has been done on Multithread models like M:1, 1:1 and M:N. Earlier M: N was considered best among all because of its performance benchmarks. Currently the Linux and Solaris have shifted from M: N to 1:1 thread model despite of the advantages of M:N. Our emphasis will be on 1:1 Model semantics for shifting. This paper gives a thorough review of operating system thread models. We have researched factors which compelled different OS platforms to shift to 1: 1 Model. Pros and cons of hybrid model and 1:1 are provided, later performance evaluation is provided. At the end of the paper some semantics are presented for preferred thread model o justify our support for 1:1 Model.

Keywords*:* **multithreading, hybrid threading, 1:1 Thread Models kernel threading, multithreading models.**

1 Introduction

Multithreading is important feature for any operating system and its importance has been increased since the emergence of multicore processors and optimized system memory architectures. A thread of execution is the smallest series of programmed instructions that can be managed independently. There are multiple threads which may exist within the same process that share memory and other resources of the process [8]. Three models are commonly used for mapping. The first methodology, M: 1 implements all threads in user space where all application software execute and appear to the kernel as single-threaded process. One of the greater drawbacks however is that it cannot get advantage from multiprocessor computers also no more than one thread is scheduled at the same time [3].

In contrast to M:N , 1:1 model which means that threads created by the user are in 1-1 mapping with threads in the kernel level. Initially Win32 API used this method. Linux platform uses C library to implement this model. The same approach is used by Solaris, NetBSD and FreeBSD. But this approach is dependent on underlying kernel [3] [8]. M:N is a hybrid model and simply a compromise between kernel-level ("1:1") and user-level ("N:1") threading. This methodology also provides significant performance when threads in an application are synchronizing with each other. As M:N provides many advantage over other two models but still it is not a feasible solution [12]. Finally, the costs for maintaining the additional code necessary for an M-on-N implementation cannot be deserted [4].

Section II will cover multithreading background, and architectures, Section III discuss all three multithreading models, Section IV will see some pros and cons of M:N and 1:1 thread models. In Section V we will focus semantics. Finally Section VI will cover performance evaluation; Section VII mentions covers Conclusion.

2 Multithreading Background and Architecture

Multithreading refers to the ability of an Operating System to support multiple and concurrent paths of execution within a process [13]. To discriminate processes from threads, the unit of scheduling and dispatching is called a thread or LWP (lightweight process) LWP is seen as virtual processor by thread library, while the unit having assigned all resources referred to as a process (or task). In a multithreaded environment, a process is defined as the unit of resource allocation and a unit of protection [9] [10].

In multithreaded environment, there is still a single PCB (process control block) and user address space for each process but there are separate stacks for each thread and also a separate control block for each thread containing register values, priority,

and other thread-related information [9] [10]. The multithreading architecture of solaris is depicted in figure 1**.** Process in figure represents a 1:1 scheduling model.

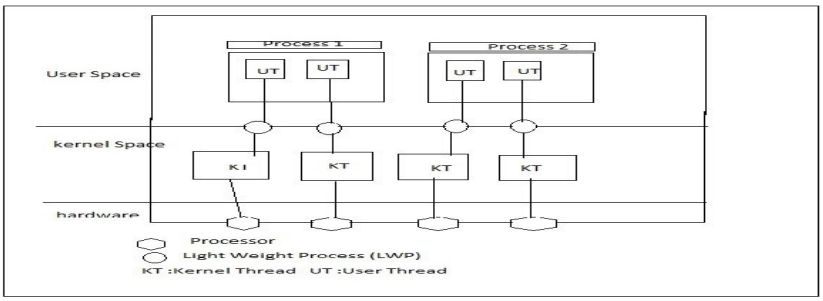

Fig. 1. Solaris Multithreading model

3 Multithreading Models

The multithreading techniques are divided into three main categories: user-threads, kernel-threads and hybrid threading models. Table 2.0 given below shows a matrix for Models, important factors and the operating system.

Table 1. Important Factors for Thread Models [2] [6] [7] [10] [11]

Model	Important factors		OS
1:N	Many user level threads to one kernel thread. Fast context switching. More efficient.	Only one user-thread in a kernel-thread can be running at a time. Blocking call will block all user threads.	Netscape and Java achieve. MRI Ruby 1.8.7 has green threads of Solaris.
M:N	Many user level threads to many kernel level threads. Allows creating sufficient number of kernel threads.Cheap creation, execution, and cleanup.	Need scheduler in user land and kernel to work with each other. Scheduling over-head increases.	IRIX, HP-UX, and Tru64 UNIX use this model, as did Solaris prior to Solaris 9.Solaris 8 and earlier.
1:1	There is one-to-one correspondence between user threads and kernel threads. Can exploit parallelism, blocking system calls. Less response time. Set individual thread affinity. Single Process ID ,Inter-thread synchronization	Thread creation involves LWP (Light weight Process) creation. Each thread takes kernel resources. Limiting the number of threads. More context switch cost	Solaris Linux. OS/2, Windows 2000, Windows NT. Linux and Windows from 95 to XP im-plemented the one-to-one model for multi-threading.

4 M:N thread model vs. 1:1 thread model

This section will cover these two models comprehensively with pros and cons of both threading models M:N (hybrid threading) and 1:1(kernel threading) [2] [5] [6] [7] [8]. There are many advantage and disadvantages of 1:1 and M:N thread models. While M:N model offers many advantages over 1:1 and M:1 model, it's not an entirely satisfactory solution[12].

Table 2. M:N thread model vs. 1:1 thread model

M:N model: Pros and Cons	1:1 model: Pros and Cons
Need for multiple schedulers: Scheduler Activation: In case of hybrid threading two schedulers are at work. Scheduler activation is implemented in kernel. So we have concluded that scheduler activation helps but not a com-plete solution [1] [5] [6].	**Simplicity:** This is simple to implement at the library level, but it's expensive in terms of kernel resources. But this method depends on underlying kernel thread model...

Number of kernel threads: How to determine the best number of kernel threads? • User specified • OS dynamically adjusts number depending on system load	**Kernel routines can be multithreaded:** The underlying kernel can be multithreaded and can simultaneously schedule many threads of the process on many processors. Blocking is done on a thread level.
Context switch cost: Context switching cost is less than 1:1 as there are two schedulers in work.	**Context switch cost:** Context switching cost is a little bit higher because the kernel must do the switching. For example, running a user code, time-slice expires, change to kernel mode to perform thread scheduling.
Priority issue: Priority is another reason that hybrid models are considered substandard. Kernel thread having high priority mapped on a low-priority user-thread would run in preference to a low-priority kernel-thread mapped on a high-priority user-thread.	**Single Process ID:** Many problems of both functionality and performance are solved by collapsing the kernel identification of all threads to a single Process ID (PID). Resource usage is attached to this single PID, making the system view of a multithreaded application accurate [15].
CPU Affinity: The thread need to execute on same processor as main process. Main process is an actual task for which threads are working. We can set the affinity but not individual thread affinity [14].	**CPU Affinity:** Operating systems that support 1:1 model has its own CPU affinity settings. Individual threads are assigned to different CPUs [14].
Duplication of scheduling supports: One of the other problems is that all scheduling supports always exist in kernel scheduler. Therefore to get the scheduling done at user-level these supports have to be duplicated at user-level	**Handling number of threads**: Operating System must scale well with increasing number of threads to use this methodology effectively. Windows and Linux tackle this difficulty well.
No true parallelism: There is no true parallelism because many user level threads are scheduled on many kernel threads	**True parallelism:** When one thread blocks the kernel simply deschedule it and pick another thread of the kernel level run queue.
Signal handling: It is done by either dedicated signal thread or upcall mechanism which is an overhead.	**Signal handling in kernel**: Signal handling is now performed within the kernel for the process, which solves the problems with POSIX signal handling.

5 Semantics of Linux and Solaris from M:N to 1:1 Model

Solaris now uses a 1:1 (user-kernel) thread model in preference to the historic M:N implementation. By simplifying the underlying thread implementation, existing applications now without requiring recompilation can see remarkable performance and stability improvements [5]. Based on performance studies and extensive workload analyses done for large enterprise customer workloads with very large numbers of processors, the M:N model was replaced with a 1:1 model starting with Solaris 9 [5].Solaris has a well-off range of process scheduling features, and some of this is reflected in the threading model. (For example, Solaris Pthreads has a priority attribute).

5.1 Signal handling

Final signal delivery at user level has many drawbacks. The signal should not be delivered to a thread that is not expecting to receive signal [4].To stop unwanted results from system calls, the system call wrappers have to be extended but it will create extra overhead. There are two ways for the signal delivery:

Dedicated signal handler thread

Signals are delivered to dedicated thread with no extra code. The drawbacks include the signal serialization and costs for the extra thread. All signals have to be direct through dedicated thread in any way even if it handles signals to other threads. This is different from the intention of the POSIX signal model which allows parallel signal handling [4].

Up-call mechanism

The other way of delivering signal at user level is by using up-call mechanism. This is what used in scheduler activation solution. This increase cost and complexity because a second signal delivery mechanism have to be implement and some support at user level is needed [4]. The alternative of all these ways is POSIX signal handling that can be done in kernel. Kernel will solve all issues related to signal handling and implementation will be simple and straightforward. Signal will be sent only to unblocked thread so there will be no unnecessary interruptions due to signals. The kernel can also select best

thread to deliver signal. This all can be possible with 1:1 thread model. The TLS patch facilitates the implementation of the 1:1 threading model without limiting the number of threads. The previous method had **limits the number of threads per process to 8192**[4].

5.2 POSIX Compatibility

Compatibility with the latest POSIX standard and to achieve source code compatibility with other platforms was the major reason of this shift.

5.3 Efficient use of Symmetric Multiprocessing(SMP)

One of the basic goals of using 1:1 thread model is to provide means to use the capabilities of multi-processor systems. Splitting the work in as many parts as there are processors can increase speedup linearly.

5.4 Low thread creation cost

Creating new thread even for small piece of work becomes low cost. ***Clone ()*** call to optimized thread creation. With use of this it is no more a heavyweight task.

5.5 Hardware/Software scalability

Increasing number of processors will not increase administrative cost and thread implementation will run sufficiently on large number of processors. Similarly there will be no limit on number of threads and it will solve problem of user application to run in separate execution context.

5.6 Maintenance of extra code cost

The cost of maintaining extra code for M:N thread model cannot be ignored at all. Especially for highly complicated code and there is lot of work needed for better implementation and code of M:N model.

5.7 Thread ID compatibility issue

With M:N thread model compatibility issue arises with other POSIX thread implementations because of each thread having a different process ID. This is really a debateable point since signals can't be used well. But using 1:1 model solves many performance and functionality issues by assigning single process ID to all threads.

Now it is easy to guess that performance can be achieved if there are same numbers of user and kernel level threads [2] [6] [8].

6 Performance Evaluation

We have tested the performance evaluation parameters on the given pseudo code by using Ubuntu 12.04 and kernel version 3.10 in POSIX thread library. These parameters are Single Process ID, CPU Affinity, Number of kernel threads and True parallelism [2][16][17]. Here results show that 1:1 Model is much better in terms of performance as compared to M:N Model

Table 3. Performance evaluation

Pseudocode Tested for performance under POSIX for LINUX using 1:1 Threading Model	

Begin
Enter choice for calculator
If choice == 1
  pthread_attr_init (&attr)
  pthread_create(&tid,&attr,add,choice)
  pthread_join (tid, NULL)
Else if choice == 2
  pthread_create(&tid,&attr,sub,choice
	Else if choice == 3
	  pthread_create(&tid,&attr,mul,choice)
	  print result
	Else if choice == 4
	  pthread_create(&tid,&attr,div,choice)
	Else
	  Print choice is invlaid
	End

Table 4 below is summarizing important and distinct benefits of implementing 1:1 multithreading model.

Table 4. Important Points for Justifying 1:1 Thread Model

Points to prefer 1:1 thread model
✓ Simplicity
✓ Kernel routines can be multi threaded.
✓ True Parallelism
✓ Inter-Thread synchronization
✓ Thread Creation is less costly
✓ Signal handling in kernel
✓ CPU Affinity cab be set by assigning different thread different CPUs.
✓ Single Process ID solves many problems of performance and functionality.
✓ Compatible with latest POSIX and other platforms.
✓ Efficient Use of SMP

Table 5.Performannce Evaluation Table for Code in Table 3.

Results for Selected parameters				
No of Processes	No. of Threads	No. of Kernel	Process ID	CPU Affinity
01	04 (02) threads per core	04 for each thread	4516	**sched_setaffinity** (*pid_t pid, size_t cpusetsize, const cpu_set_t *cpuset*) This method is used by 1:1 model for setting affinity

7 Conclusion and Future Work

This paper focuses the multithreading models in detail. Key factors of both models have been analyzed we have given reasons why Solaris and Linux shifted to 1:1 implementation and retired the historical M:N thread model.

Solaris and Linux have many reasons of this semantics mainly discussed in this paper are scheduler activation, signal handling, number of threads, CPU, Affinity settings, inter-thread synchronization, Single Process ID, scalability and performance improvement without recompilation. 1:1 thread model is proven best in later versions of Solaris and Linux as there is no performance penalty.

In future more work can be done to simulate the idea with some implementation in a controlled environment. For this purpose Linux thread API and JAVA threads could be used. Further it can be worked out how 1:1 Model data structures impact on multicore processor's performance.

References

1. Neil Brown "A Many-to-Many Threading Model for Multi-core Architectures" Communicating Process Architectures 2007 Alistair A. McEwan, Steve Schneider, Wilson I fill, and Peter Welch, IOS Press, (2007)
2. A technical white paper ©2002 Sun Microsystems, Inc. "Multithreading in the Solaris[a] Operating Environment".
3. Dave McCracken, Ottawa Linux Symposium 2002 "POSIX Threads and the Linux Kernel" IBM® Linux® Technology Center Austin, TX.(2002)
4. Drepper, U., & Molnar, I., "THE NATIVE POSIX THREAD LIBRARY FOR LINUX." Red Hat.(2003)
5. An Oracle Technical White Paper ,"Red Hat Enterprise Linux to Oracle Solaris Porting Guide" ,(2012)
6. Xiao-Feng Li "On runtime technology and programming languages", http://xiao-feng.blogspot.com/2008/08/thread-mapping-11-vs-mn.html
7. Tei-Wei Kuo "Chapter 4: Multithreading", National Taiwan University, (2005)
8. "Threads" , http://en.wikipedia.org/ wiki/ Thread_ (computing).
9. Kevin Haghighat, "Multithreading", (2008)
10. W. Stalling, Operating Systems: Internals and Design Principles, 7th edition, Prentice-Hall, (2012)
11. Silberschatz, A. and Galvin, P. B., Operating System Concepts, 8th edition, Addison-Wesley, (2012)
12. Bryan M. Cantrill "Runtime performance analysis of M-to-N scheduling model", Department of Computer Science, Brown University, (1996)
13. A.Frank-P.Weisber, "Operating System – Threads Implementation", BIU, (1999)
14. Mike Anderson "Understanding and Using SMP/Multi-core processors", The PTR Group, Inc. (2008)
15. L. Blunt Jackson, "NPTL: The New Implementation of Threads for Linux" (2005)
16. Edward A. Lee, "The problem with Threads", Electrical Engineering and Computer Sciences University of California at Berkeley, (2006)
17. Anupam Chanda, Khaled Elmeleegy, Romer Gil, Sumit Mittal, Alan L. Cox, and Willy Zwaenepoel, "An Efficient Threading Model to Boost Server Performance", Rice University, Dept. of Computer Science, Switzerland.