Multithreading in the Kylin Operating System for High End Computing

Zhang Yingxing Wu Qingbo
School of Computer, National University of Defense Technology,
Changsha, Hunan, P.R.China
willy_young@hotmail.com, wqb123@263.net

Abstract

This paper presents the architectural and implementation details of multithreading in kylin operating system. This system provides a foundation for efficient and flexible threads on both uniprocessor and multiprocessor machines .The work is based on the scheduler activations kernel interface proposed by Anderson et al. [1] for user-level control of parallelism in the presence of multiprogramming and multiprocessing . Preliminary results on a SMP enterprise server demonstrate that the implementation is very efficient.

1. Introduction

The need for high-end systems has led to new interest in high-end computing research, development, and procurement. The current trend in large-scale HEC systems is to leverage operating systems developed for other areas of computing. An informal survey of the most powerful HEC systems currently deployed (http://www.top500.org) reveals that all but a few of these machines run commodity operating systems that were not specifically designed for large scale, parallel computing platforms. However, we note that the Sandia's ASCI/Red and the Cray T3E have demonstrated the highest level of scaling efficiency for several complex scientific applications and that both of these platforms employed specialized operating systems on their compute nodes. It is therefore important that research in the development of specialized operating systems be augmented by research that seeks to consolidate key features of specialized operating systems with main-stream, commodity operating systems.

Kylin operating system is focused on high performance, availability and security, and was funded by a Chinese government-sponsored 863 High-Tech R&D program. It has been organized in a hierarchy model, including the basic kernel layer which is similar to Mach, the system service layer which is similar to BSD and the desktop environment which is similar to Windows. It has been designed to comply with the UNIX standards and is compatible with Linux binaries. In order to support parallel applications efficiently, we have emphasized on the multithreading mechanism.

The purpose of this paper is to describe the design and implementation of multithreading in kylin. First , as motivation , Section 2 describes traditional thread implementations .In Section 3 we discuss the architectural and implementation details of the multithreading in kylin. Section 5 presents performance results and Section 6 concludes the paper.

2. Background and Related Work

Multithreading is a popular programming and execution model that allows multiple threads to exist within the context of a single process, sharing the process's resources but able to execute independently. The threaded programming model provides developers with a useful abstraction of concurrent execution. Threading implementations typically have components in both user and kernel space. It is possible to do everything from the one side or the other, but each approach has problems. With everything on the user side, all related threads are part of one single process (which can only run on one CPU at a time), and multi-processor systems are underutilized. With everything on the kernel side, the kernel scheduler must bear a heavy load. Approaches have ranged from the 1:1 pure kernel thread model in which each user thread has its own kernel thread, to the M:1 model in which the kernel sees only one normal process, with an arbitrary number of threads with which to schedule in user space. The M:N model falls in between, associating M user threads with each of N kernel threads.

2.1 M:1 model (ala FreeBSD's libc r)

In M:1 model ,which is also called useland threading ,userland threads are implemented entirely in an application program, with no explicit support from the kernel. In most cases, a threading library is linked in to

Proceedings of the Eighth International Conference on High-Performance Computing in Asia-Pacific Region (HPCASIA'05)
0-7695-2486-9/05 $20.00 © 2005 IEEE

the application, though it is also possible to hand code user threading within an application.

The libc_r use a combination of a timer signal (SIGALARM for libc r) to allows the userland threads scheduler (UTS) to run, and *setjmp*()/*longjmp*() calls to switch between threads. For avoiding blocking in a system call. ,libc_r converting potentially blocking system calls to non-blocking. This works well in all cases except for operations on so-called fast devices such as local filesystems, where it is not possible to convert to a non-blocking system call. libc r handles non-blocking and incomplete system calls by converting file descriptors to non-blocking, issuing I/O requests, then adding the descriptors to a central *poll*()-based event loop.

Userland threads have the advantage of being very fast in the simple case, though the complexities of call conversion eats into this performance advantage for applications that make many system calls.

2.2 1:1 model (ala Linux's LinuxThreads)

The 1:1 model is confusingly referred to as Process-based threading much of the time. Process-based threads are threads that are based on some number of processes that share their address space, and are scheduled as normal processes by the kernel. LinuxThreads implements process-based threading.

The Linux kernel uses the clone() function to create new processes such that they share the address space of the existing processes. Flags control parent/child resource sharing, where resources range from everything (memory, signal handlers, file descriptors, etc.) to nothing. While the usual fork() inherits resources from the parent, it may share nothing. Copy-on-write techniques ensure each process gets its own copy as soon as either one tries to modify a shared resource.

Programs can call the clone() function as a system call, using it directly to produce multithreaded programs. However, it is completely Linux-specific and non-portable. Since there is no external standard, there is no guarantee that its interface will be stable. Threading library implementations do in fact use the clone() system call, and it is the job of library maintainers to keep up with kernel changes.

Process-based threading also has some inherent performance and scalability issues that cannot be overcome: Switching between threads is a very expensive operation and Each thread (process) requires all the kernel resources typically associated with a process.

2.3 M:N model (ala Solaris Multithreading)

The M:N model ,which is also called Multi-level threading , is a hybrid of user-level and process-based threading. The idea of multi-level threading is to achieve the performance of userland threading and the SMP scalability of process-based threading. Ideally, most thread scheduling is done by a UTS to avoid the context switch overhead of kernel calls, but multiple threads can run concurrently by running on more than one process at the same time.

In practice, multi-level threading' s main shortcoming is its complexity. Just as Solaris uses LWPs to address the POSIX compliance issues mentioned above for purely process-based threading. And, in theory, LWPs are light-weight, though Solaris's LWPs no longer generally meet this criterion.

The overhead of the multi-level scheduling is also worse than excepted.

2.4 Scheduler Activations

The scheduler activations (SAs) as presented in *Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism*[1], and is meant only as a basis for the more complete treatment of multithreading in kylin .

SAs differ from multi-level scheduling in that additional kernel facilities are added in order to provide the UTS with exactly the information and support it needs in order to control scheduling. Simply put, SAs allow the kernel and the UTS to do their jobs without any guess work as to what the other is doing.

A process that is using SAs does not have a kernel stack or PCB. Instead, every time a process is run, a SA is created that contains a kernel stack and thread control block (TCB), and the process runs in the context of the SA. When the SA is preempted or blocked, machine state is stored in the SA's TCB, and the kernel stack is optionally used for completion of a pending system call.

3 Multithreading in Kylin

3.1 Kylin threads Architecture

The multithreading mechanism in kylin implemented in M:N model strives to merge the advantages of userland and process based threading while avoiding the disadvantages of both approaches .The following figure shows the architecture:

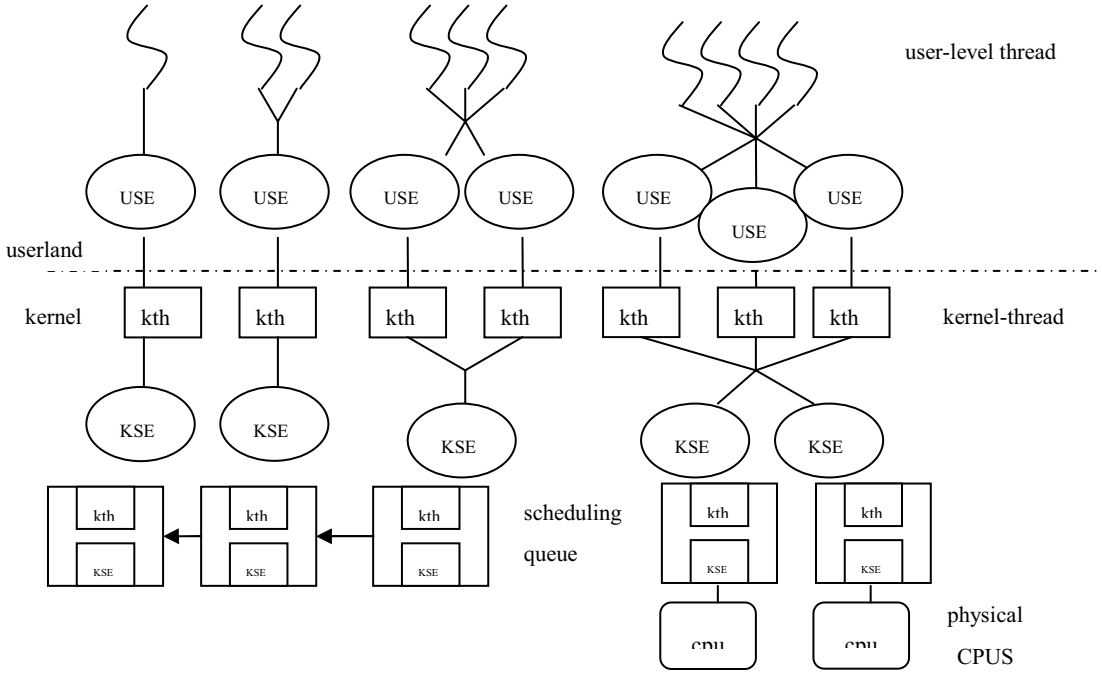

Figure 1. The architecture of multithreading in kylin

At the user-level, kylin provides USE structures which serve as an implicit indication of the kernel processor allocation to the user-level threads library. USEs are mapped one-to-one to kernel threads, which are the actual kernel entities for multithreading the same address space. The user-level library can obtain the handle of a kernel thread indirectly, by scheduling a user-level thread onto a USE.

In the kernel, each process has several KSE structures which serve as virtual processors. The kernel thread will be put into scheduling queue, when it has been bound to a KSE in the same process. In this way ,one process' maximum degree of parallelism is associated with the number of the KSE.

3.2 Kylin Multithreading Implementation

In this section , we discuss the implementation details of multithreading in kylin . Section 3.2.1 discusses the implementation of Userland-scheduled entities and Kernel- scheduled entities further in detail. Section 3.2.2 describes the communication mechanism between kernel and userland. Section 3.2.3 describes the implementation of UTS .

3.2.1 Userland-scheduled entities and Kernel-scheduled entities

Userland-scheduled entities(USEs) and Kernel-scheduled entities (KSEs) are based in concept on scheduler activations, which are treated in detail in the *SA* paper [1] .The Figure.1 shows how kernel thread and user-level thread interact by using USEs and KSEs.

KSEs are used in the kernel in the scheduler queues, and as a general handle. Given a pointer to a KSE, it is possible to access its associated process, kernel thread and the thread's USE instances. A USE is simply the user-level representatives of kernel threads , It is possible to access the structure named use_thr_mailbox which contains the state of a suspended thread of execution, by giving a pointer to the thread's USE . When a running thread is blocked, the execution state is saved in the use_thr_mailbox so that when it is possible to continue execution, the thread can be re-attached to a KSE and continued. When a thread is preempted, the execution state is saved in a use_thr_mailbox so that an userland state can be handed to the UTS. When the UTS loads the context and starts executing the thread, it also sets its curthread pointer to the thread's mailbox. This allows the kernel (who knows where the KSE mailbox is) to know where to store the return context for the thread should it become required.

KSEs are only evident within the kernel. The interface with userland only deals with USEs. KSEs themselves are irrelevant to userland because they serve essentially as an anonymous handle that binds the various kernel structures together.

All the processes start out with one kernel thread ,one USE and one KSE, and its concurrency level is one. Programs can create new user-level threads and kernel threads by using POSIX interface . As new kernel threads are created, the count of USE will increase to keep bijection to the kernel thread ,and the count of KSE can be adjusted by calling kse_create syscall ,up to the maximum number of CPUs available for the process.

3.2.2 Communication Mechanism

Kylin operating system use upcall mechanism to notify the kernel event to userland thread scheduler , which is also evolve from the SA paper in concept .

At the time upcalls are activated via a system call, program flow changes radically to run userland threads scheduler .

There are three types of kernel events will cause the program to receive upcalls from the kernel : a process's degree of parallelism increases and the process adds a new KSE, current thread enters kernel and blocks , current thread wants to return userland when there are completed thread in the same process.

After the kse_create() syscall is called, a new KSE structure and a new upcall structure are created. The upcall has been given a pointer to a mailbox in userland, as well as the addresses of a userland stack and function, that it can use to perform upcalls should it need to . In kylin operating system ,the function is always the UTS. Then kernel bind the upcall structure up with the standby thread which attaches to current thread , and put the standby thread into scheduling queue. When the standby thread has been scheduled on CPU , it will call the function supplied on the given stack ,which is always the UTS.

If the thread enters the kernel and blocks, it will be disconnected from the KSE and a new thread attached. The standby thread of the blocked thread will be bound up with a upcall structure ,then enters the UTS, which realizes that the thread has blocked and looks for more work to do. After the UTS find another runnable thread if the new thread in turn enters that kernel, then there is a chance that the previous syscall for the other thread has completed. If so , the final return context for the original thread will have been written into its mailbox context area. The thread will have been released, so that userland thread which becomes the completed thread now is only in userland again . In this way , UTS will be invoked as quickly as possible when thread blocks.

When current thread wants to return userland and there are completed threads in the same process , kylin turn it into the upcall instead of releasing it . But before to do, kylin will put the completed threads into the scheduling queue of userland , then the UTS can follow this queue to decide which thread to run next .

3.2.3 Userland thread scheduler and kernel scheduler

The KSE-based UTS is actually simpler than is possible for a userland-only threads implementation, mainly because there is no need to perform call conversion. The following is a simply representation of the core UTS logic :

1. Find the highest priority thread in the process . Optionally heuristically try to improve cache locality by running a thread that may still be partially warm in the processor cache.
2. Set a timer that will indicate the end of the scheduling quantum.
3. Run the thread.

The UTS always has the information it needs to make fully informed scheduling decisions, but there are some circumstances that can cause temporary scheduling inversions, where a thread may continue to run to the end of its quantum despite there being a higher priority runnable thread . This can happen when:

1. A new thread is created that has a lower priority than its creator, but a higher priority than a thread that is concurrently running on another processor.
2. A running thread(labeled as A) is preempted by the kernel, and the upcall notification causes preemption of another thread(labeled as B) , which is higher priority than thread A , though thread C is also running on another processor and has a lower priority than both A and B. In this case, A will be scheduled and C will continue to run, even though thread B is higher priority than C .

Kylin don't resolve this problem for the two reasons:

1. Solutions to this problem require additional system calls in which the UTS explicitly asks the kernel to preempt KSEs. This is expensive.
2. Temporary inversions are logically equivalent to the race condition where the UTS determines that a thread should be preempted in favor of scheduling another thread, while the thread races to complete its quantum. It is not important whether the UTS or the thread wins the race.

Two chief characteristics of the kernel scheduler are mandatory in order to support KSEs :

1. The pairs of thread and KSE are are placed in the scheduler queues, rather than processes. This enables concurrent execution of KSEs that are associated with the same process.
2. The state for blocked (incomplete) system calls is stored in thread structures . This means that this queue consists of thread rather than pairs. In other words, the scheduler deals with pairs in some places, and threads only in other places .

4 . Evaluation

In this section, we present performance evaluation results of multithreading in kylin . The experiments were conducted on LangChao NF420R server with four Intel 2.2G Hz P4 processors which support Hyper-Threading technology and 1G DDR memory. By modifying BOIS and kernel configuration ,we simulate UP and 8 processors SMP hardware environment.

The evaluation is split in two parts . We evaluate the performance of low-level primitives like thread creation and context switching overhead first. Then ,we will present preliminary results of our system on the multiprogramming .

The following Figure illustrates the cost of creating and deleting a thread on the multithreading system in kylin ,comparing with lib_r lthread library running on FreeBSD and the linuxthread library which are also supported by kylin .

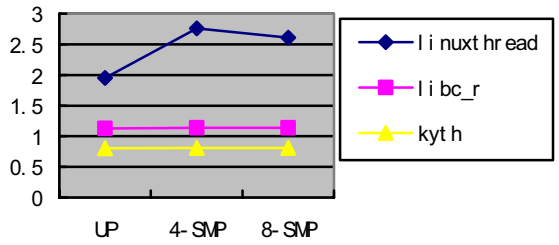

Figure 2. Cost of thread creation and delete

The LMbench benchmark forks a number of child threads and then joins with them . The child thread immediately returns . In each iteration the program forks10 threads . This loop runs 100 times ,and then the LMbench gets the average value of the cost .The figure is drawn in logarithmic scale and show that the cost of creating threads on kylin is much less than the costs of performing the same opertations on linuxthread library , and is close to the cost on libc_r.

LinuxThreads implements process-based threading . When the linuxthread library creats a new thread , a new processes is created such that it shares the address space of the existing process .So the cost of thread creation is almost close to the cost of process creation .And all the thread creations in the library will be controlled by one thread named manage thread .There is another performance bottleneck of this library. libc_r library implements userland threading . Userland threads are implemented entirely in an application program, with no explicit support from the kernel. Its performance of thread creation is certainly better than the performance of linuxthread library.

The kylin multithreading system implements M: N model. When a process creates a new thread for the first time , it will create several kernel threads once . The default number is according to the degree of parallelism . Then the process only creates userland thread after the first time . The performance of this mechanism will be close to the performance of userland threading when the operate of creating thread is frequency . And with the good garbage collection mechanism, our performance is even better than the performance of libc_r library.

Figure 3,4and 5 show the cost of context switching on different hardware . We use the LMbench which executes several threads that pass the tokens in turns .The number of threads is changed from 50 to 200 for simulating different work load . There are also three implementations compared, one with linuxthread library , one with libc_r library and one with kyth library .

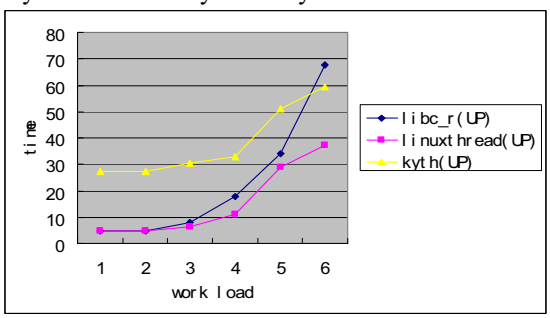

Figure 3. 50 threads

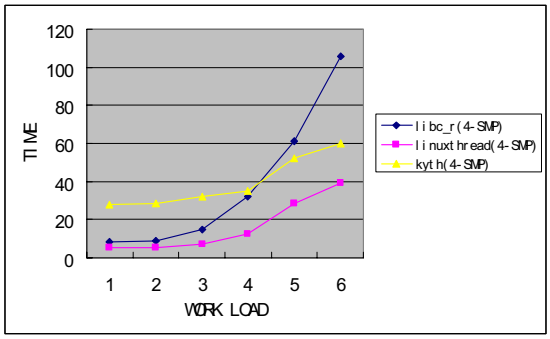

Figure 4. 100 threads

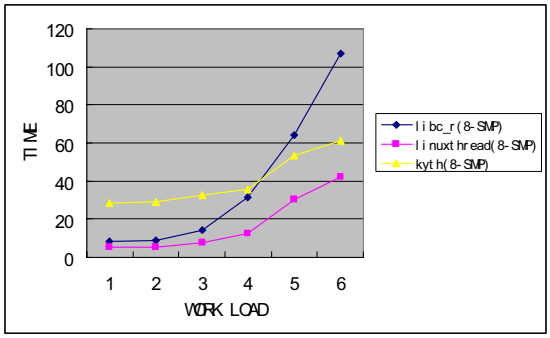

Figure 5. 200 threads

As the implement of userland threading ,the performance of libc_r library gets very low with the heavy load .The performance of process-based threading is much higher than the former .

The cost of kyth library is a bit higher than the other two libraries at light work load, because the M:N thread model increases more communication between kernel and userland .The advantage of kyth is that its performance is insensitive to work load ,its performance under heavy work load is higher than libc_r library, and gets close to linuxthread library.

In the second part , we use SpecWeb benchmark to test the performance of kyth thread library and

linuxthread thread library.Table.1 shows the result .

Table 1.

Multithreading	menu	w/o perl	with perl
linuxpthread	Bandwidth(b/s)	69393	17174.7
	Response time (ms/op)	1743.2	7064.7
kyth	Bandwidth(b/s)	176964	20723
	Response time (ms/op)	678.5	5771
kyth's speedup ratio	Bandwidth	+155.0%	+20.7%
	Response time	-61.1%	-18.3%
remark	500 cons	res.178, res.179	res.191, res.190

5. Conclusion

This paper has presented the implementation details of multithreading in kylin operating system based on the scheduler activations model . The goal of the implementation was to provide fine-grain parallelization and multiprogramming scalability at affordable runtime costs. We implement a kernel mechanism that allows the kernel and userland to support threaded processes and communicate with each other effectively so that the necessary information is available for both to do their jobs efficiently and correctly. Measurements were taken that demonstrate thread performance on our system. This project continues to evolve, and the future goal is clear: implementation of higher performance .

References

[1] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism", ACM Transactions on Computer Systems, Vol. 10, No. 1, February 1992, Pages 53-79.

[2] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, "The Design and Implimentation of the 4.4 BSD Operating System". Addison-Wesley, 1996.

[3] Paul Barton-Davis, Dylan McNamee, Raj Vaswani, and Edward D. Lazowska, "Adding scheduler activations to Mach 3.0. Technical Report 3", Department of Computer Science and Engineering, University of Washington, August 1992.

[4]Brown and M. Seltzer, "Operating system benchmarking in the wake of lmbench: A case study of the performance of NetBSD on the Intel x86 architecture.", Proceedings of the 1997 ACM SIGETRICS Conference on Measurement and Modeling of Computer Systems, pages 214–224, 1997.

[5]Christopher Small and Margo Seltzer , "Scheduler activations on BSD: Sharing thread management between kernel and application. Technical Report 31", Harvard University, 1995

[6]N. J. Williams , "An implementation of scheduler activations on the netbsd operating system" , USENIX Annual Technical Conference, 2002.

[7]T. Garfinkel , "Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools." , Proceedings of the Symposium on Network and Distributed Systems Security (SNDSS), pages 163–176, February 2003.

[8]M.Haines, "On designing lightweight threads for substract software " , USENIX 1997 Annual Technical Conference . Anaheim, CA ,pages 243-255 ,jan ,1997

[9] D. Craig, "An Integrated Kernel-Level and User-Level Paradigm for Efficient Multiprogramming", CSRD Technical Report No. 1533, University of Illinois at Urbana-Champaign, October 1998.

[10] Jason Evans, "Kernel-Scheduled Entities for FreeBSD", http://www.aims.net.au/chris/kse/docbook/

[11]W. Richard Sstevens ,Advanced Programming in the UNIX Environment,Addison Wesley Publishing Company

[12] Steve Kleiman, Devang Shah, and Bart Smaalders, *Programming with Threads*, SunSoft Press