Title:	Lesser Bear - a Light-weight Process Library for SMP Computers -

Authors:	Hisashi Oguma and Yasuichi Nakayama
Affiliation:	The University of Electro-Communications
Contact Author:	Hisashi Oguma
Full address:	c/o Prof. Y. Nakayama, Department of Computer Science, The University of Electro-Communications, 1-5-1 Chofugaoka, Chofu, Tokyo 182-8585 JAPAN
Phone number:	+81-424-43-8072
Fax number:	+81-424-43-5334
E-mail:	oguma-h@igo.cs.uec.ac.jp

Lesser Bear — a Light-weight Process Library for SMP computers —

Hisashi Oguma and Yasuichi Nakayama
Department of Computer Science
The University of Electro-Communications
Chofu, Tokyo 182-8585, JAPAN
oguma-h@igo.cs.uec.ac.jp

Abstract We have designed and implemented a light-weight process (thread) library called "Lesser Bear" for SMP computers. Lesser Bear has high portability and thread-level parallelism. Creating UNIX processes as virtual processors and a memory-mapped file as a huge shared-memory space enables Lesser Bear to execute threads in parallel.

Lesser Bear requires exclusive operation between peer virtual processors, and treats a shared-memory space as a critical section for synchronization of threads. Therefore, thread functions of the previous Lesser Bear are serialized.

In this paper, we present a scheduling mechanism to execute thread functions in parallel. In the design of the proposed mechanism, we divide the entire shared-memory space into partial spaces for virtual processors, and prepare two queues (Protect Queue and Waiver Queue) for each partial space. We adopt an algorithm in which lock operations are not necessary for enqueueing. This algorithm allows us to propose a scheduling mechanism that can reduce the scheduling overhead. The mechanism is applied to Lesser Bear and evaluated by experimental results.

Keywords: thread library, SMP computer, parallelism, scheduler design

1 Introduction

Recently, multiprocessor systems have become popular, as is illustrated by the widespread use of PC-based multiprocessors. Therefore, many UNIX-compatible operating systems support symmetric multiprocessor (SMP) computers. Systems that effectively utilize the feature of SMP computers are required. In particular, a light-weight process, sometimes called a thread, is attracting much attention for its use as a basic processing unit. In order to effectively utilize SMP computers, we have developed a thread library, called "Lesser Bear," for SMP computers. Lesser Bear has following features:

- high portability; and
- thread-level parallelism.

Thread libraries are utilized for parallel applications because thread management (e.g. switching between peer threads, creation, etc.) is more inexpensive than that of the UNIX process. But if there are fine-grain threads in an application, thread management takes place frequently, and this overhead influences the turnaround time of the application. For example, in fork-join type applications, thread synchronization takes place frequently. Therefore, the more fine-grain the threads are, the more frequently thread management will occur.

Lesser Bear creates some UNIX processes inside the application as virtual processors in order to execute each thread in parallel. Therefore, the previous Lesser Bear design required an exclusive function between multiple virtual processors for thread scheduling. If a fine-grain application, in which context switching occurs frequently, the context switching for each virtual processor is serialized inside. Consequently, the previous design prevented the thread scheduler from running in parallel.

In this paper we propose the design and implementation of a scheduling mechanism. This mechanism has the following features:

- scheduling thread in parallel inside of Lesser Bear; and
- Low-overhead scheduling.

For scheduling threads in parallel, the thread scheduler needs to run on each virtual processor.

In previous design, Lesser Bear stored all the thread-contexts in a huge shared-memory space where every virtual processor can access uniformly. In this paper, we divide the huge shared-memory space equally for every virtual processor and provide two queues ("Protect Queue" and "Waiver Queue") for each divided space. Each virtual processor manages a provided space. Protect Queue is handled without lock operation because only an assigned virtual processor, referred to as an owner, enqueues into the Protect Queue and dequeues from it. In Waiver Queue, only the owner enqueues into that queue but any virtual processor can dequeue from it. Consequently, the enqueue method for the Waiver Queue requires no lock operation. Since the implemented scheduling mechanism requires no lock operation in enqueueing, Lesser Bear is able to reduce the scheduling overhead.

In this paper, we evaluate the implemented scheduling mechanism on an SMP computer with 8 CPUs. Experimental results show that we achieve scheduling threads in parallel with low-overhead.

The reminder of the paper is organized as follows. Section 2 presents related works and overview of Lesser Bear. Section 3 presents the proposed scheduling mechanism for improving Lesser Bear. Section 4 presents the experimental results of improved Lesser Bear. The final section concludes the paper.

2 Former Threads Libraries

This section discusses works related to the thread library and Lesser Bear's features.

2.1 Related Work

In general, threads can be implemented as:

- an implementation that requires some modifications in a kernel (e.g. Scheduler Activations [3]); or
- a library implementation (e.g. PTL [2]).

A kernel implementation can construct a suitable system for the architecture, but makes the system less portable. However, a library implementation, called a thread library, is not dependent on the architecture and operating system (OS).

A variety of thread libraries have been developed [3, 2, 4, 5, 6, 7]. But existing thread libraries suffer from one or both of the following problems:

- lack of portability;
- thread-level parallelism.

Most of the existing thread libraries have only one virtual processor. Therefore, there is no parallelism at the thread.

2.2 Overview of Lesser Bear

To utilize the advantages of the thread library and SMP computer, we have designed and implemented a thread library, called Lesser Bear [8]. Figure 1 shows a diagram of Lesser Bear. Lesser Bear has two features; high portability and thread-level parallelism.

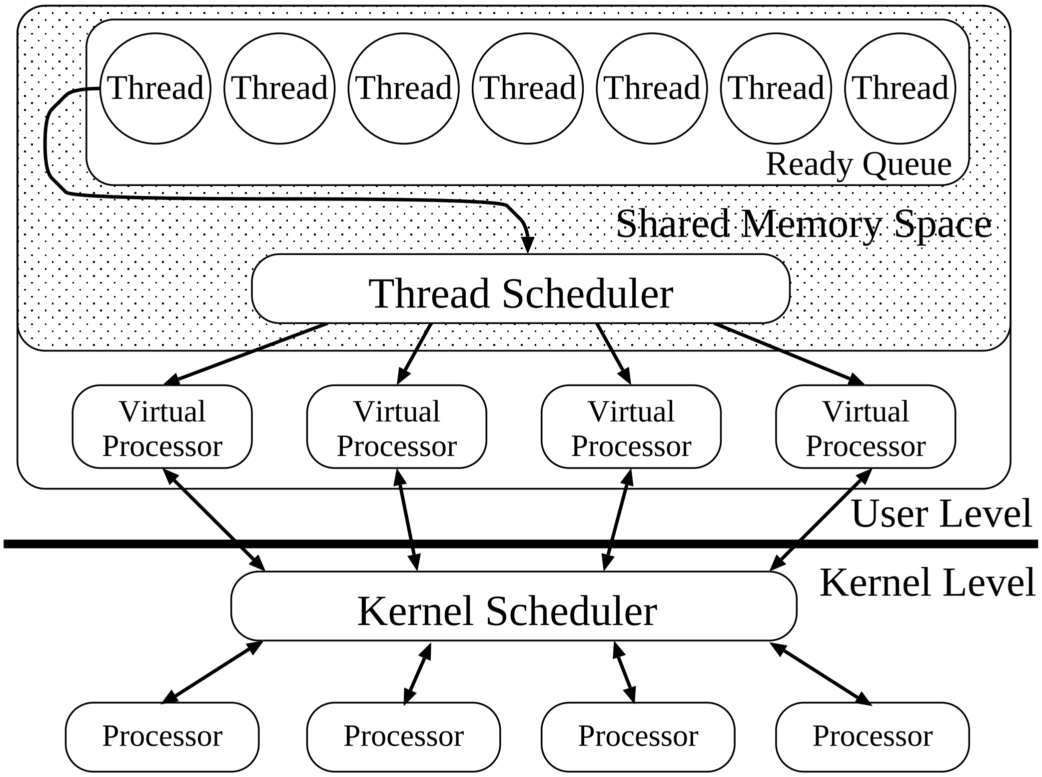

Figure 1: Our thread library model.

Most of the previous thread libraries contain one virtual processor to deal with threads. Consequently, they have no parallelism at the thread. To satisfy thread-level parallelism, Lesser Bear creates some UNIX processes as virtual processors. LinuxThreads [5] and PPL [7] have multiple virtual processors and satisfy thread-level parallelism. Creating some virtual processors enables a thread library to satisfy thread-level parallelism. However, each virtual processor can not deal with arbitrary threads in this situation.

Moreover, a thread library requires a function that suspends and resumes a running thread. Therefore, a thread library requires space to store all the thread-contexts that include personal data (e.g. stack, register set).

In Lesser Bear, all the thread-contexts are stored in a shared-memory space where every virtual processor can access uniformly. Consequently, any virtual processor can deal with any thread.

UNIX processes are assigned to CPUs and run concurrently in order to run an application linking Lesser Bear on an SMP computer. For this reason, thread-level parallelism is satisfied in Lesser Bear. And Lesser Bear initially creates a memory space shared with all virtual processors that is as large as possible.

We require the following features for Lesser Bear:

- portability;
- context switching by user-level interval timer;
- a standard user interface;
- a huge shared-memory space; and
- exclusive operation between peer virtual processors.

Table 1: Operating systems on which Lesser Bear works as designed.

OS	types	feature
SunOS 4.1.4	BSD UNIX	Uni-processor
SunOS 5.5.1	SVR4 UNIX	SMP
FreeBSD 2.2.8	BSD UNIX	Uni-processor
FreeBSD 3.0	BSD UNIX	SMP
Linux 2.0	SVR4 UNIX	SMP
IRIX 6.4.1	SVR4 UNIX	SMP

In the rest of this section, we will describe in detail the portability and a huge shared-memory space.

2.3 Portability

In general, it is required that libraries are not dependent on the architecture and OS. To satisfy this demand, we have implemented Lesser Bear using C language and standard UNIX libraries. For context switching, we have adopted `setjmp()` and `longjmp()` to achieve portability.

Lesser Bear requires an OS that can run multiple processes in parallel and provide the memory-mapped file system.

Table 1 presents operating systems that Lesser Bear can run. Lesser Bear has only two or three lines of implemented source codes that depend on the OS. By using this feature, we expect that Lesser Bear will also run easily on other architectures.

2.4 Huge Shared Memory Space

For each virtual processor dealing with any thread, Lesser Bear creates a huge shared-memory space (the order of 1 GB). All data structures (e.g. thread-contexts, Ready Queue) are stored in this space. A thread-context includes personal data (e.g.

stack pointer, register environment). To store a large amount of thread-contexts, a huge shared-memory space is necessary. In Lesser Bear, a huge shared-memory space is implemented by a memory-mapped file.

These strategies assure that Lesser Bear has both high portability and thread-level parallelism.

3 Design of Scheduling Mechanism

In this section, we describe the problems the previous Lesser Bear design, and propose a scheduling mechanism to solve the problems.

3.1 Problems of Scheduler Serialization

In Lesser Bear, data structures (e.g. Ready Queue, thread-contexts, thread table) are stored in a shared-memory space, and each virtual processor can execute any thread in parallel. Therefore, the entire shared-memory space has been treated as a critical section, and all virtual processors are required to operate separately.

When a virtual processor has entered a critical section and the entire shared-memory space is locked, the other virtual processors are suspended [9]. Consequently, thread management (e.g. context switching, scheduling) is serialized (Figure 2).

The more processors an SMP computer has, the more frequently this phenomenon occurs. As Figure 3 shows, when a virtual processor is switching a thread-context, other virtual processors are prevented from switching it. For this reason, every thread management is serialized in the previous Lesser Bear design.

To solve the serialization problem, the following two solutions are considered [1].

- A critical section is divided for each data structure, and each one is provided with lock variables.

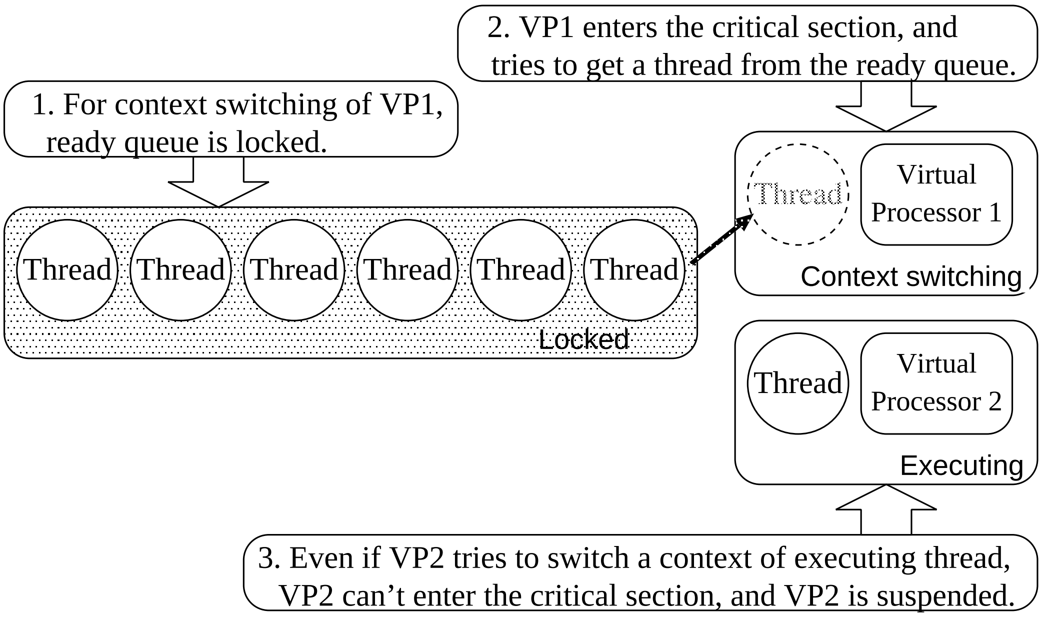

Figure 2: Blocking of virtual processor.

- The entire data structure is divided for each virtual processor, and each one has a partial data structure in a local space.

For the first solution, multiple lock variables are required to manage partial critical sections, so that the structure of thread library tends to be complex. Moreover, because multiple lock variables are required for thread management primitives (e.g. thread creation, context switching), deadlock occurs easily.

For the second solution, exclusive operation among all virtual processors is required to keep the shared data consistent. However, if the other virtual processors do not affect the data stored in each local space, lock operation is not necessary.

We adopt the second solution. In this way, all the thread-contexts are stored in a shared memory-space. Each virtual processor has thread identifiers for executing threads. Therefore, it is not necessary that each local space makes very large. If a virtual processor has been idle, it imports executable threads from other virtual

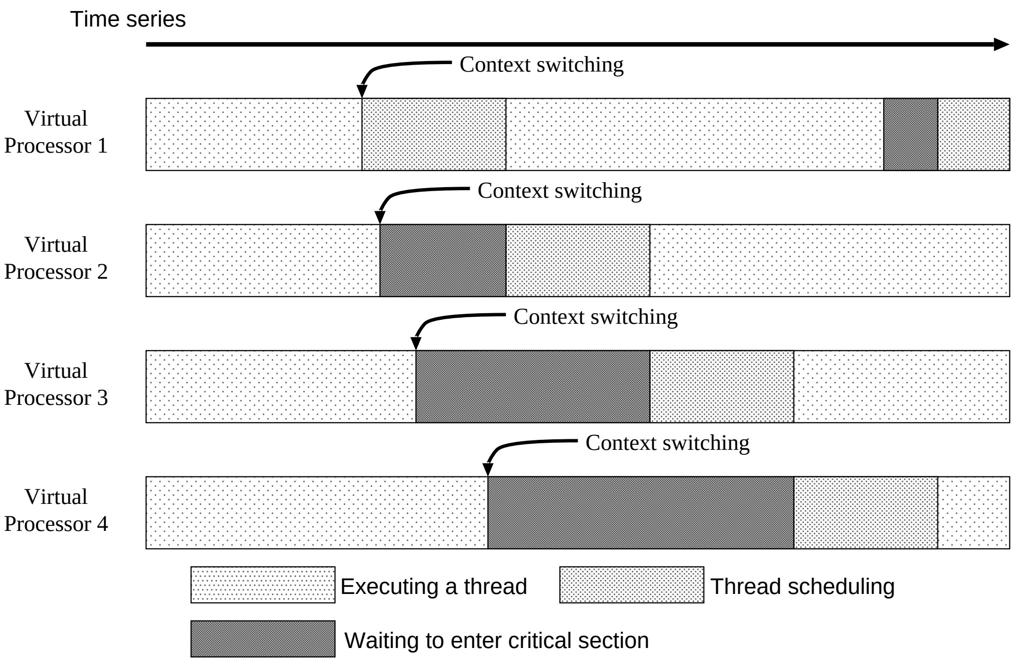

Figure 3: An example of blocking a virtual processor on SMP computers.

processors. Exclusive operation is only utilized for thread movement.

3.2 New Scheduling Mechanism

In the scheduling mechanism presented in this paper, a entire shared-memory space is divided among virtual processors, and a local queue is prepared in each partial space. However, we propose two queues ("Protect Queue" and "Waiver Queue") instead of the local queue. Each virtual processor manages its own space and is supplied with two queues. Figure 4 shows our proposal for scheduling threads in parallel.

Protect queue only allows the owner to enqueue and dequeue. Therefore, the owner does not have to use any lock operations. In switching the thread-context, the owner of Protect Queue removes a thread from the head of Protect Queue. For load balancing, the capacity of each Protect Queue is always uniform between every

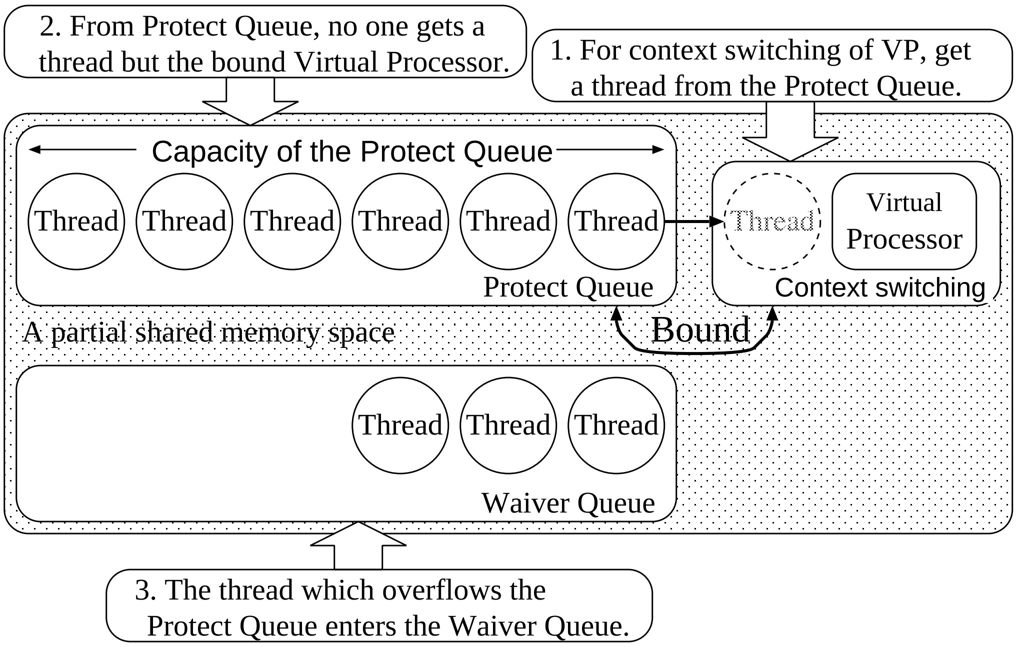

Figure 4: A new scheduler design for running itself in parallel.

Protect Queue. If Protect Queue overflows, the owner adds the thread to the end of the Waiver Queue.

Waiver queue allows the owner to enqueue, but it lets everyone dequeue. In removing a thread from Waiver Queue, lock operation among virtual processors is necessary. Thus, virtual processors can not remove a thread simultaneously from Waiver Queue.

It has been reported that no lock operation is required when only one virtual processor is permitted to enqueue and only one (not necessarily the same) virtual processor is permitted to dequeue. [10]. Consequently, lock operation is not necessary for adding a thread to Waiver Queue (Figure 5).

Enqueueing to the queues is frequent in Lesser Bear, so that reducing overhead in enqueueing is related to the effective utilization of the system. Note that the synchronization among the virtual processors trying to remove threads from Waiver

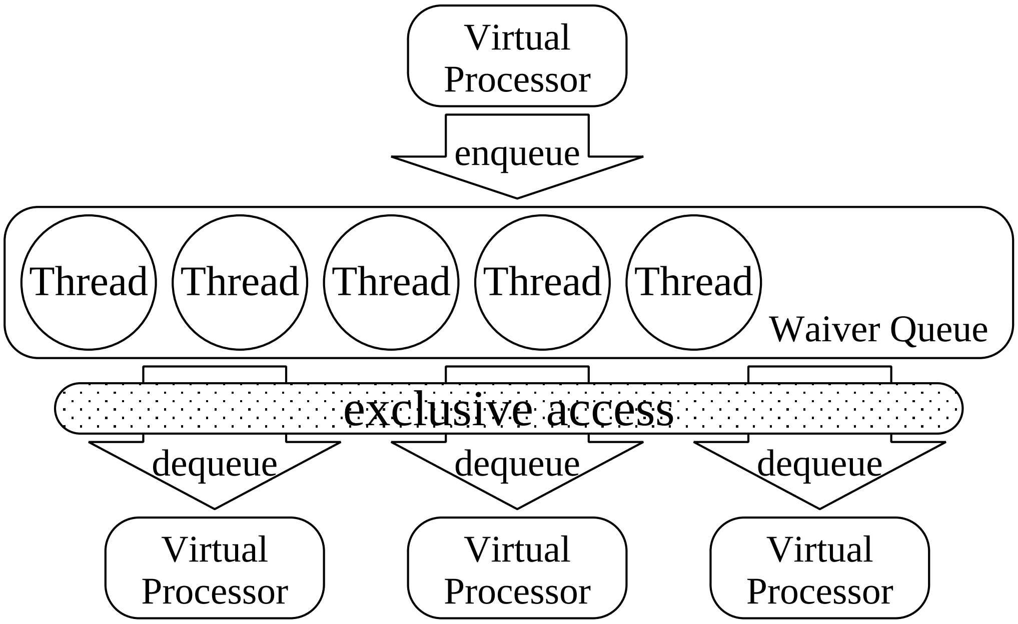

Figure 5: Operations about Waiver Queue (enqueueing and dequeueing).

Queue is required. Since idle virtual processors dequeue from Waiver Queue, the overhead does not influence the performance.

We also apply a similar mechanism to the waiting queue for mutex (mutual execution) variable operations. One virtual processor that releases a mutex variable handles the waiting queue without lock operation. These techniques enable Lesser Bear to reduce the lock operation.

The scheduling mechanism proposed in this paper is able to reduce the lock operation more than the mechanism in the previous Lesser Bear design. Consequently, the overhead of thread management in the scheduling threads is reduced.

4 Experimental Result

In this section, we describe the evaluation of designed scheduling mechanism in comparison with the previous Lesser Bear design. All experiments are conducted on a Sun microsystems SPARC Server 1000 running version 5.5.1 of the SunOS. This system has a single-bus shared-memory architecture and is equipped with eight SuperSPARC processors, running at a clock rate of 40 MHz. The system has 640 MB of physical memory.

4.1 Performance Evaluation for Scheduler

In order to evaluate the scheduling mechanism, we compare the proposed and previous Lesser Bear design.

We first measure the cost for thread scheduling. In this experiment, we let Lesser Bear have one virtual processor.

Table 2: Scheduler design performance.

	former design	proposed design
Costs of running scheduler (μsec)	144.4	83.2

Table 2 compares the scheduling cost of the previous and proposed designs. For this experiment, we utilize an application in which two threads are created. One thread yields the virtual processor to the other and scheduler repeats the context switching for a long time.

The result shows that the scheduling cost of the previous design contains the cost of lock operations. In this experimental environment, the cost of lock operation is 65 microseconds. The combined cost of the proposed design and the lock operation equals the cost of the previous design.

Next, we count the number of times for thread scheduling per second in each virtual processor. For this experiment, we create 128 threads running 10 minutes in the application program. The time quantum of thread is 10 milliseconds. Lesser Bear adopts a semaphore to synchronize the virtual processors. Semaphore operation allows a process that is the waiting in the queue to be put into semaphore sleep. If there is no ready process in the OS, the CPU becomes idle. In order to effective utilize the experimental platform in this experiment, we create 16 virtual processors, twice as many as the number of CPUs.

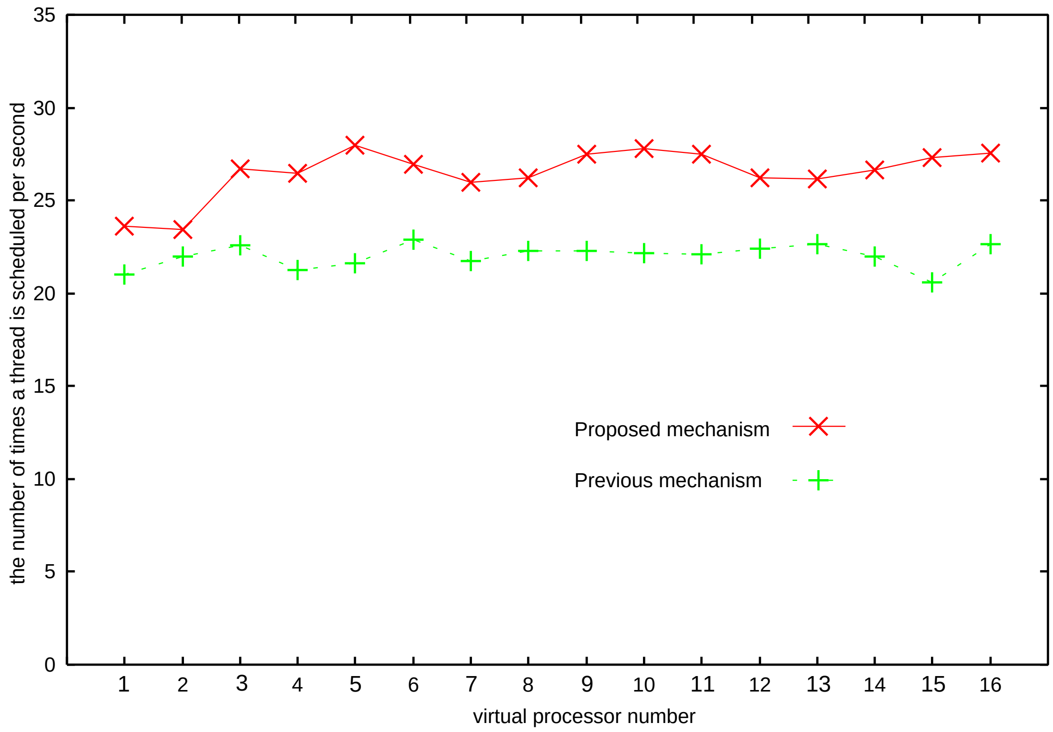

Figure 6: The number of times a thread is scheduled.

In figure 6, the horizontal axis of the graph represents the number of times a thread is scheduled per second. The vertical axis represents the virtual processor number. Figure 6 shows that the proposed mechanism enables the internal scheduler to run frequently. This is partly because the lock operation is rarely necessary to schedule thread.

4.2 Performance Evaluation using Application Programs

In this section, we run an application program in order to present the advantages of the proposed scheduling mechanism. We adopt the radix sort as an application program for this experiment.

The radix sorting algorithm [11] treats keys as multidigit numbers, in which each digit is an integer with a value in the range $\{0 \cdots (m-1)\}$, where m is the radix. Radix sort works by breaking keys into digits and sorting one digit at a time, starting with the last digit. For efficiency, m often becomes the value of 2 raised to the power nth. By distributing all keys, it is easy to execute radix sort program in parallel, and we can expect to achieve high scalability.

When the radix is 4, we separate a set all of the keys for each thread and sort each thread in order to parallelize radix sort algorithm by thread programming as follows (Figure 7):

1. Count the number of keys on each element (0, 1, 2 and 3).
2. From the result of 1, merge all elements from all threads.
3. From the result of 1, create the partial sum of all elements until the previous thread.
4. From the above results, determine the offsets for each element.
5. Transfer the keys indicated by the offsets.

For this strategy, we require barrier synchronization for merging and transferring.

Pthread [12], which is adopted for the interface of Lesser Bear, does not support the barrier synchronization. In this experiment, we implement the barrier synchronization by utilizing mutex variables and condition variables.

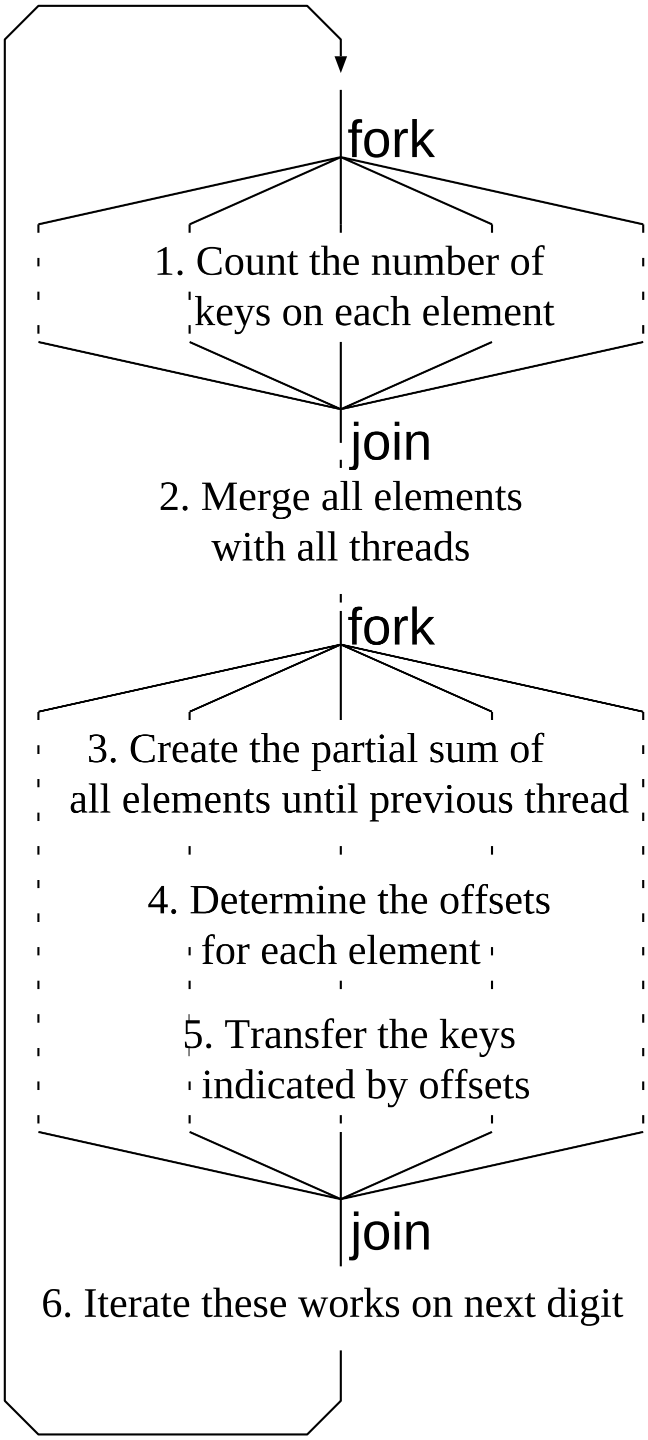

Figure 7: The model of parallelize radix sort program.

At first, we compare the performance of the scheduling mechanism of the previous and proposed designs. In the application, the number of keys is 2^{22}, and 2^8 threads are created. In this experiment, we vary the size of the radix (2^1, 2^2, $\cdots$, 2^8), and measure the turnaround time. In Figure 8, the horizontal axis of the graph represents the size of the radix, and the vertical axis represents execution time normalized to that of the previous scheduling mechanism.

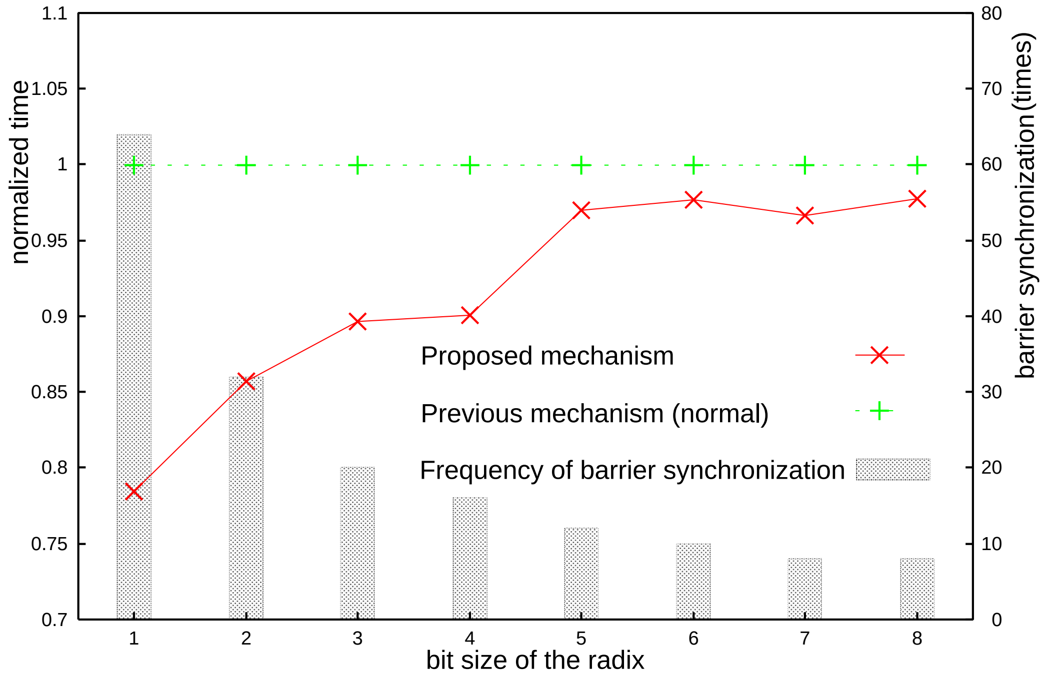

Figure 8: Comparison of turnaround time between previous and proposed Lesser Bear designs.

The lesser radix size is, the more fine-grain the fork-part becomes and the more frequent barrier synchronization occurs. Therefore, thread management (e.g. mutex variable control, condition variable control) happens frequent and it becomes system overhead.

When the radix sort program is run on the proposed scheduling mechanism, the lesser the radix size is, the better the performance of the proposed scheduling mechanism in comparison with the previous mechanism. This means that there can be a lot of thread management even when there is a small radix. The lock operation is not necessary for thread management in the proposed scheduling mechanism, so that the proposed mechanism performs well with a small radix. When radix is 2^1, the barrier synchronization is generated 64 times and turnaround time is reduced about 22 %.

Next, we compare Solaris threads and Lesser Bear in which the proposed scheduling mechanism design is implemented.

Solaris threads is a thread library supported by SunOS 5.x. Solaris threads is a kernel implementation, so that Solaris threads yields the best performance on SunOS 5.x.

But, Solaris threads requires kernel support for thread managements, so that we can not expect good performance in an application in which thread management occurs.

Figure 9 shows a comparison with Solaris threads and the proposed Lesser Bear with the scheduling mechanism. The horizontal axis of the graph represents the radix size. The vertical axis represents the turnaround time. The radix sort program is as the same as that used in the previous experiment.

When radix size is small, fork-join operation occurs frequently, and the number of serial parts increases inside of the application. Figure 9 shows the application features.

Figure 9 shows that Lesser Bear has good performance when the radix is small. This is mainly because the thread management's overhead, especially mutex variable control and condition variable control, of Solaris threads is high. In Lesser Bear, the algorithm in **3.2** is utilized for mutex variable control, and this reduces the control overhead.

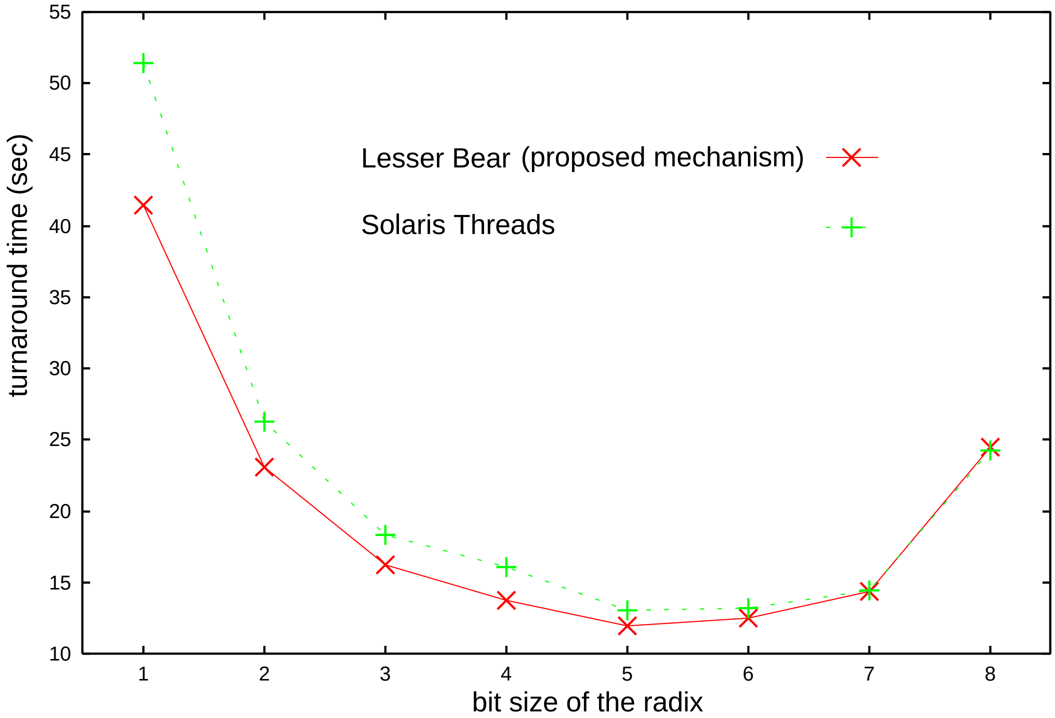

Figure 9: Comparison of turnaround time between the proposed designed Lesser Bear and Solaris threads.

These results show that proposed scheduling mechanism achieves thread scheduling on each virtual processor in parallel and reduces the thread management overhead.

5 Conclusions

In this paper, we have proposed a scheduling mechanism to schedule threads in parallel in each virtual processor and to reduce scheduling overhead.

To accomplish thread scheduling in parallel, we divide a huge shared-memory space among virtual processors and provide two queues (Protect Queue and Waiver Queue) for each divided space.

Protect Queue requires no lock operation for enqueueing and dequeueing, because it does not allow anyone but the owner to enqueue and dequeue. Waiver queue allows only the owner to enqueue, but it lets everyone dequeue. In this paper, for enqueueing and dequeueing, we adopt an algorithm in which no lock operation is necessary to enqueue. The implementation of this algorithm enables Lesser Bear to reduce the scheduling overhead.

In the experiments, we show the effectiveness of reducing overhead in thread scheduling, and show the scheduling thread in parallel on each virtual processor. We have adopted the radix sort program as the application program.

From the results of running the application, we have confirmed that the overhead of thread management in the proposed scheduling mechanism is lower than the overheads of previous Lesser Bear and Solaris threads.

References

[1] Anderson TE, Lazowska ED, Levy HM. The Performance implications of Thread Management Alternatives for Shared-Memory Multiprocessors. *IEEE*

Transactions on Computers 1989; **38**(12):1631–1644.

[2] Abe K. PTL – Portable Thread Library. http://www.media.osaka-cu.ac.jp/~k-abe/PTL/.

[3] Anderson TE, Bershad BN, Lazowska ED, Levy HM. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems* 1992; **10**(1):53–79.

[4] Mueller F. A Library Implementation of POSIX Threads under UNIX. *Proceedings of the Winter 1993 USENIX Conference*, January 1993; 29–41.

[5] Leroy X. Linuxthreads – POSIX 1003.1c kernel threads for Linux. http://pauillac.inria.fr/~xleroy/linuxthreads/.

[6] Provenzano C. Pthreads version 1_70. http://www.mit.edu:8001/~proven/pthreads.html.

[7] Sakamoto C, Miyazaki T, Kuwayama M, Saisho K, Fukuda A. Design and Implementation of Parallel Pthread Library (PPL) with Parallelism and Portability. *Systems and Computers in Japan* 1998; **29**(2):28–35.

[8] Oguma H, Nakayama Y. Lesser Bear — a Light-weight Process Library for SMP computers —. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada, USA (PDPTA '2000)*, June 2000; 2451–2457.

[9] Kaieda A, Nakayama Y, Tanaka A, Horikawa T, Kurasugi T, Kino I. Analysis and Measurement of the Effect of Kernel Locks in SMP Systems, *Concurrency: Practice and Experience*; to appear.

[10] Suzuki M, Watanabe T. A Lightweight Solution for the Producer-consumer Problem, *Report CS 00-05*, Department of Computer Science, University of Electro-Communications, October 2000.

[11] Zagha M, Blelloch GE. Radix Sort for Vector Multiprocessors. *Proceedings of the 1991 conference on Supercomputing '91*, November 1991; 712–721.

[12] Nichols B, Buttlar D, Farrell JP. *Pthreads Programming.* O'Reilly & Associates; 1996.