Title: Lesser Bear: a lightweight process library
for SMP computers - scheduling mechanism
without a lock operation

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 lightweight process library for SMP computers – scheduling mechanism without a lock operation

Hisashi Oguma and Yasuichi Nakayama
Department of Computer Science
The University of Electro-Communications
1-5-1 Chofugaoka, Chofu, Tokyo 182-8585 JAPAN
oguma-h@igo.cs.uec.ac.jp

Abstract We have designed and implemented a lightweight process (thread) library called 'Lesser Bear' for SMP computers. Lesser Bear has thread-level parallelism and high portability. Lesser Bear executes threads in parallel by creating UNIX processes as virtual processors and a memory-mapped file as a huge shared-memory space. To schedule thread in parallel, the shared-memory space has been divided into working spaces for each virtual processor, and a ready queue has been distributed.

However the previous version of Lesser Bear sometimes requires a lock operation for dequeueing. We therefore proposed a scheduling mechanism that does not require a lock operation. To achieve this, each divided space forms a link topology through the queue, and we use a lock-free algorithm for the queue operation. This mechanism is applied to Lesser Bear and evaluated by experimental results.

Keywords: thread library, SMP computer, parallelism, scheduler design, lock-free algorithm

1 Introduction

Recently, multiprocessor systems have become popular, as 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 lightweight process, sometimes called a thread, is attracting much attention for its use as a basic processing unit. In order to enhance the effectiveness of SMP computers, we have developed a thread library, called 'Lesser Bear'. Lesser Bear has thread-level parallelism and high portability.

Lesser Bear creates some UNIX processes inside the application as virtual processors in order to execute each thread in parallel. Lesser Bear stores all the thread-contexts in a huge shared-memory space which every virtual processor can access uniformly. Furthermore, for scheduling threads in parallel, Lesser Bear divides the huge shared-memory space equally for every virtual processor and provides 'Protect Queue' and 'Waiver Queue' for each divided space. Protect Queue allows only the owner virtual processor to enqueue and dequeue. Waiver Queue allows only the owner to enqueue, but lets everyone dequeue. We have also adopted an algorithm for Lesser Bear in which lock operations are not necessary for enqueueing. This algorithm enables Lesser Bear to reduce the scheduling overhead.

In the previous version of Lesser Bear, Protect Queue heaps some threads for its own virtual processor, making frequent lock operations unnecessary. Lock operations are not necessary for enqueueing, either. However in queueing operations, dequeueing occurs at the same frequency as enqueueing. During dequeueing, virtual processors are serialized inside and prevented from parallel processing.

We therefore propose a scheduling mechanism requiring no lock operations. The lock operation is necessary in the previous version of Lesser Bear because Waiver Queue lets everyone dequeue. To achieve the proposed mechanism, we specify a virtual processor that dequeues from the Waiver Queue. In addition to this, all the partial shared-memory spaces form link topology through the Waiver Queues. For the Waiver Queue operation, we use the same lock-free algorithm as the previous version of Lesser Bear.

Lesser Bear requires threads to move between peer virtual processors in order to keep the amount of threads for each virtual processor equal. The proposed mechanism enables threads to move among virtual processors through the Waiver Queue, which does not require a lock operation in enqueueing and dequeueing. Therefore, the overhead of thread movement does not influence the turnaround time of an application, even if threads move among virtual processors frequently. Consequently, Lesser Bear can equalize the amount of threads that each virtual processor has, can keep each virtual processor busy, and can improve CPU efficiency.

We evaluate the scheduling mechanism on an SMP computer having 8 CPUs. Experimental results show good performance when lock-free is used.

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

2 Background

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 [1]); 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, known as a thread library, is not dependent on the architecture and operating system (OS).

A variety of thread libraries have been developed [2, 3, 4, 5, 6, 7], but all suffer from one or both of

- thread-level parallelism, or
- lack of portability.

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

On the other hand, LinuxThreads [5], Solaris threads [6] and PPL [7] have thread-level parallelism. However, it is hard to say that they have high portability because

- LinuxThreads and Solaris threads only work on Linux and SunOS 5.x, respectively, and
- PPL consists of about 20 % OS dependent module [7].

2.2 Overview of Lesser Bear

To harness the advantages of using a thread library and an SMP computer together, 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: thread-level parallelism and high portability.

Most of the previous thread libraries contain only 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] also have multiple virtual processors and satisfy thread-level parallelism. All the thread-contexts are stored in a huge shared-memory space which every virtual processor can access uniformly. Lesser Bear

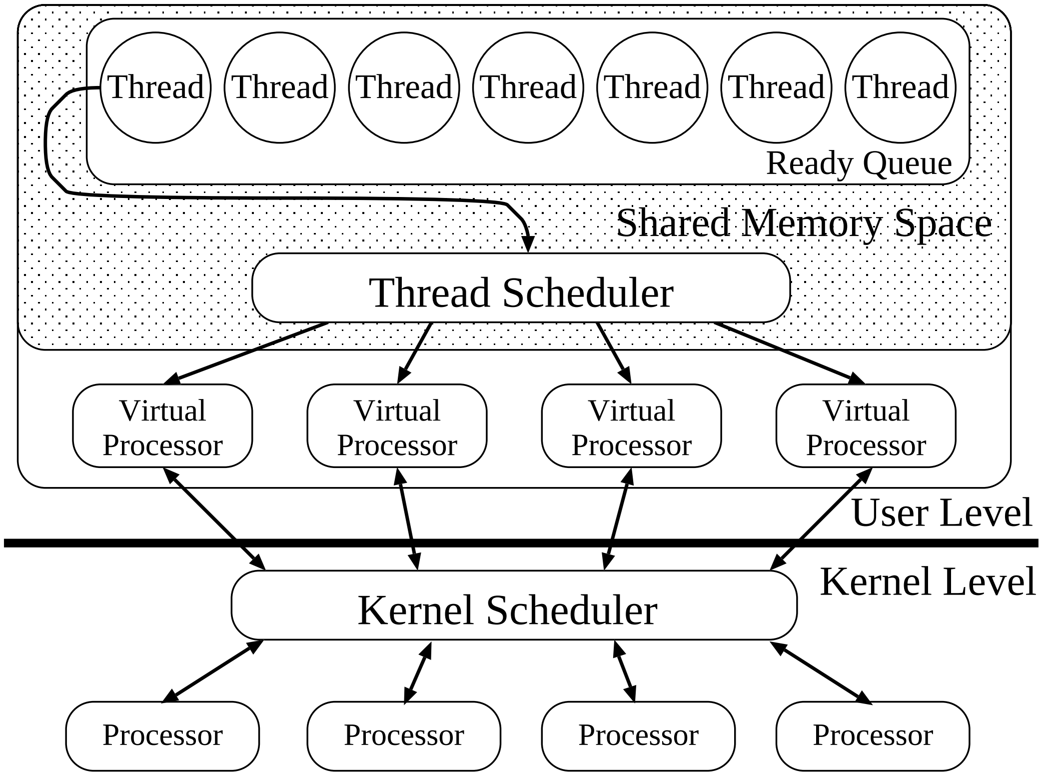

Figure 1: Our thread library model.

is implemented by using only C language and standard UNIX libraries. Table 1 presents operating systems that Lesser Bear can run. Lesser Bear has only two or three lines of implemented source codes. By using this feature, we expect that Lesser Bear will also run easily on other architectures.

Moreover, for scheduling in parallel, Lesser Bear allocates a partial shared-memory space to each virtual processor as a working space, and provides 'Protect Queue' and 'Waiver Queue' for each working space (figure 2). Each virtual processor usually manages a provided partial shared-memory space.

Protect Queue only allows an assigned virtual processor, referred to as an owner, to enqueue and dequeue, so that Protect Queue is done without a lock operation. The capacity of the Protect Queue changes dynamically but it is always uniform

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

OS	architecture	feature
SunOS 4.1.x	SPARC	Uni-processor
SunOS 5.x	SPARC	SMP
SunOS 5.x	intel	SMP
FreeBSD	intel	SMP
Linux	intel	SMP
IRIX 6.4.1	MIPS	SMP

between every Protect Queue.

Waiver Queue only allows the owner to enqueue. If the Protect Queue overflows, the owner adds the thread to the own Waiver Queue. If there is no thread in both queues, a virtual processor removes one from a Waiver Queue which another virtual processor owns and adds its own Protect Queue. Waiver Queue lets everyone dequeue, so that a lock operation among virtual processors is necessary in order to prevent a thread from being simultaneously moved from the Waiver Queue (figure 3).

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 [9]. Consequently, a lock operation is not necessary for adding a thread to the Waiver Queue.

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 accomplished in Lesser Bear. And Lesser Bear initially creates the largest possible memory space that can be shared with all virtual processors.

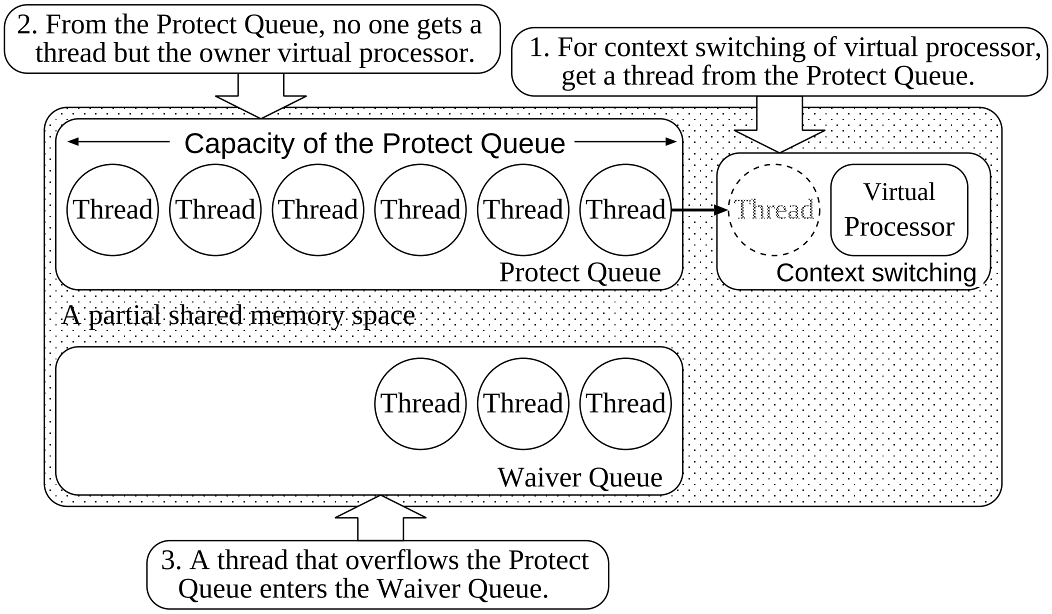

Figure 2: Protect Queue and Waiver Queue.

3 Design of New Scheduling Mechanism

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

3.1 Lock Operations of Waiver Queue

In order to use the advantages of SMP computers, we have divided an entire shared-memory space and proposed two queues to schedule threads in parallel.

We let the previous version of Lesser Bear heap some threads in the Protect Queue. While a virtual processor is holding a lock for dequeueing from the Waiver Queue, it adds as many threads as possible to its own Protect Queue. Therefore, a virtual processor does not have to use the lock operation frequently. In addition to this, a queueing technique eliminates the need for lock operations when enqueueing.

However in queueing operations, dequeueing occurs at the same frequency as

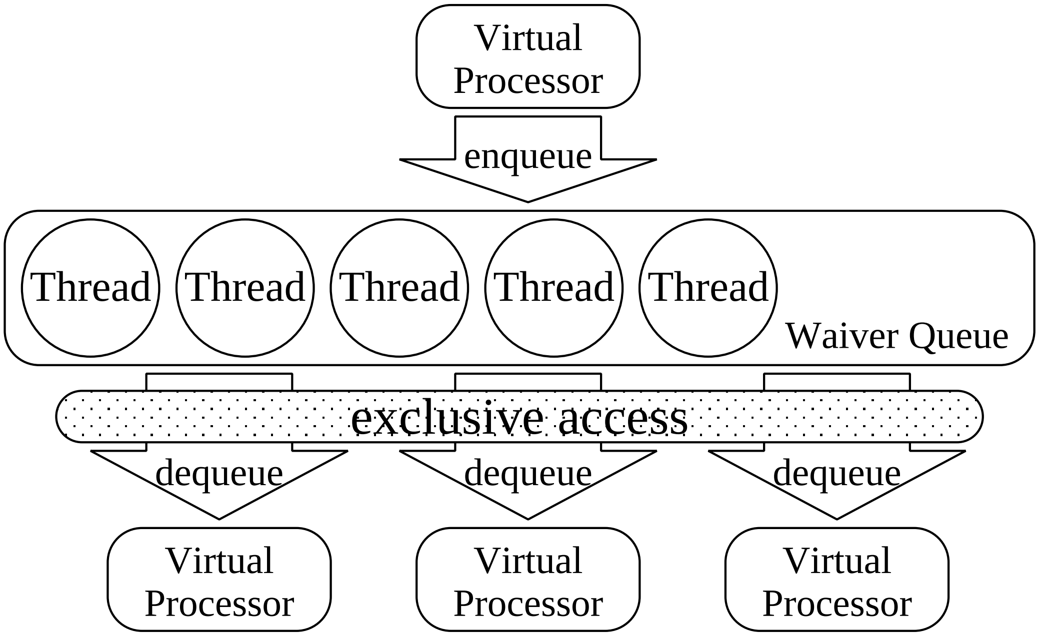

Figure 3: Insert and remove operations about the Waiver Queue.

enqueueing. When a virtual processor has no thread, it tries to obtain threads from the Waiver Queue that another virtual processor manages. Idle virtual processors could obtain threads from the Waiver Queue in parallel as far as they did not remove from same Waiver Queue. However, Waiver Queue is handled with a lock operation in the previous version of Lesser Bear, and virtual processors may serialize during dequeueing as shown in figure 4. Lesser Bear uses `semop` system call to synchronize the virtual processors to maintain high portability. The more frequently the lock operation occurs, the more frequently the system call is called, thus it reduces the performance of Lesser Bear [10].

To eliminate the lock operation in thread scheduling, two solutions are considered.

- The first solution is decreasing the cost of the lock operation. Low-cost and user-level lock operations that utilize `Test-and-Set` and `Compare-and-Swap`

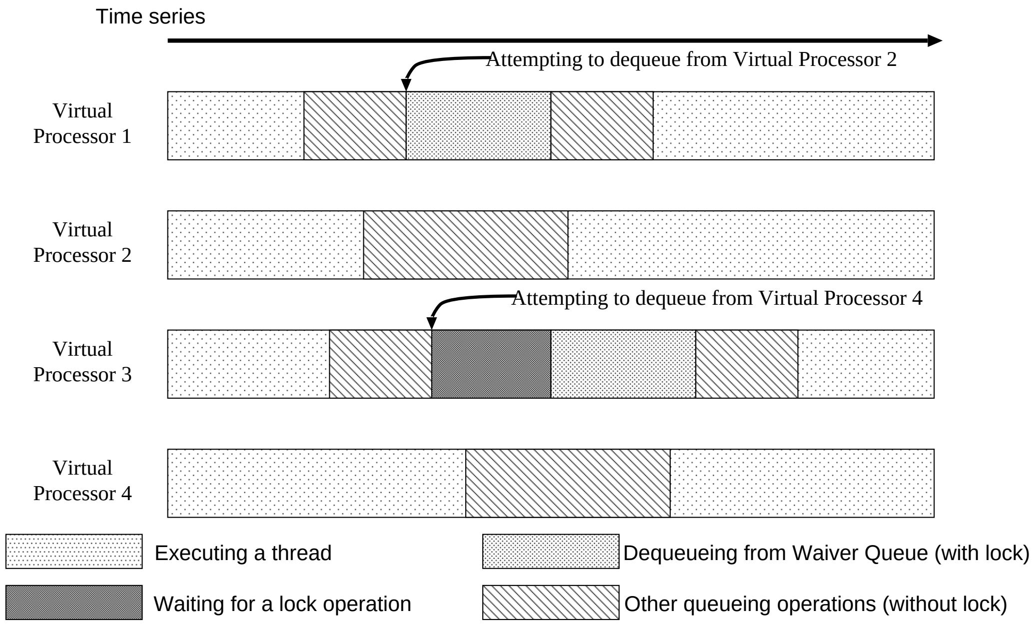

Figure 4: Example of blocking a virtual processor when dequeueing.

are implemented inside of the kernel [11, 12, 13, 14]. These depend on the specificity of the architectures. Lock-free queue operations have also been studied [15, 16]. They utilize `Test-and-Set` and `Compare-and-Swap`. Consequently, they have no portability.

- The second solution is constructing the data structure that does not require a lock operation. Moving a thread between peer virtual processors uses a shared space which they can access uniformly. For eliminating a lock operation, it is necessary to devise an access method for the space from a virtual processor.

We adopt the second solution because of maintaining high portability. In this way, we improve the queueing operation for the Waiver Queue, which is shared with multiple virtual processors. As a matter of course, the queueing operation is implemented using C language and standard UNIX libraries.

For reducing lock operations, SALSA[17] and nano-threads[18] have employed hierarchical ready queues[19] for their scheduler design. However we propose another design to eliminate a lock operation in this paper.

Next section, we describe a scheduling mechanism that requires no lock operation in detail.

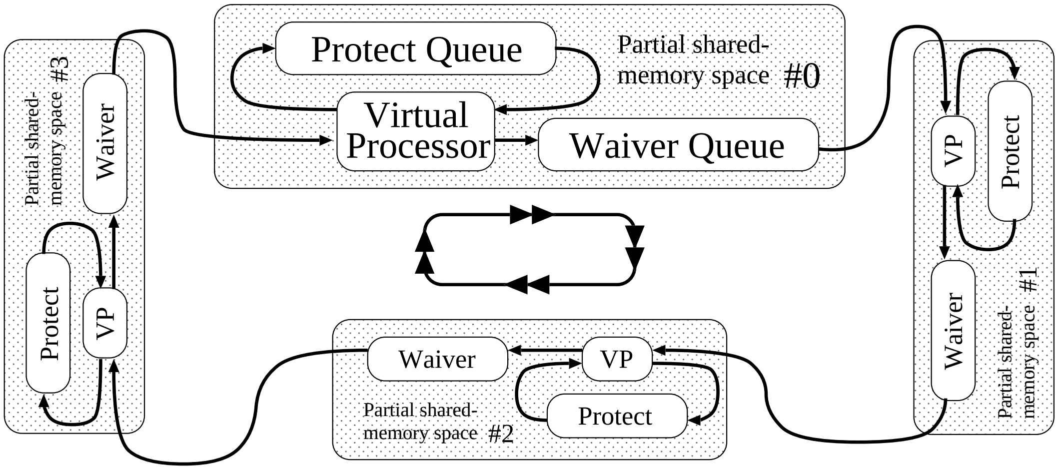

Figure 5: Diagram of rotator scheduling mechanism.

3.2 Rotator Scheduling Mechanism

Here is a description of the design and implementation of a thread scheduling mechanism having no lock operation. We use features of the Protect Queue and the Waiver Queue that are used in the previous version of Lesser Bear.

As described in section 2.2, Protect Queue and Waiver Queue have the following features for queueing operations in the previous version of Lesser Bear.

- Protect Queue is handled without a lock operation because only the owner enqueues into the Protect Queue and dequeues from it.
- Waiver Queue allows only the owner to enqueue, but lets everyone dequeue. When removing a thread from the Waiver Queue, a lock operation among virtual processors is necessary.

In this paper, we propose a rotator scheduling mechanism that is applied to these features. Figure 5 shows a diagram of the rotator scheduling mechanism which has four virtual processors.

We implement the mechanism with following process [20].

- An entire shared-memory space is divided among virtual processors like the previous version of Lesser Bear.
- Protect Queue and Waiver Queue are prepared in each partial space.
- Each partial space forms a link topology through the Waiver Queue.

Hereinafter, we refer VP_i as the ith virtual processor $(0 \leq i \leq n-1 \mid$ where n is the number of virtual processors), and we define $forward(i)$ and $backward(i)$ as follows:

$$
\begin{aligned}
forward(i) &= \begin{cases} 0 & (i = n-1) \\ i+1 & (0 \leq i \leq n-2) \end{cases} \\
backward(i) &= \begin{cases} n-1 & (i = 0) \\ i-1 & (1 \leq i \leq n-1) \end{cases}
\end{aligned}
$$

To the Protect Queue and the Waiver Queue, VP_i adds and removes a thread according to the following algorithm.

- Fundamentally, a virtual processor adds a suspended thread to the Protect Queue and removes the next thread awaiting execution from the Protect Queue.
- If a thread overflows from the Protect Queue, a virtual processor adds it to the own Waiver Queue.
- Whenever VP_i switches the thread-context, VP_i removes all threads from the Waiver Queue which $VP_{backward(i)}$ manages, and adds threads to the Protect

Queue until the limit of the Protect Queue's capacity. The remaining threads are added to the own Waiver Queue.

- If $VP_{forward(i)}$ is sleeping when inserting threads to the Waiver Queue which VP_i manages, VP_i lets $VP_{forward(i)}$ awaken.
- If there are no threads in the own Protect Queue and the Waiver Queue that $VP_{backward(i)}$ manages, VP_i falls asleep.

Lesser Bear does the above procedure whenever it is interrupted by an interval timer.

In this mechanism, different virtual processors share a Waiver Queue. However, lock operations are not necessary because one only enqueues and the other only dequeues. Therefore, moving a thread among virtual processors as well as enqueueing and dequeueing requires no lock operation. This is because a thread moves between peer virtual processors through a Waiver Queue. In the remainder of this section, we describe an advantage of the proposed mechanism.

When one virtual processor monopolizes threads, the rest become idle, and CPU utilization is reduced. Thus, it is necessary to allocate threads evenly among all virtual processors. In the previous version of Lesser Bear, an idle virtual processor removes threads from the Waiver Queues, which the other virtual processors manage. This is because removing thread from the Waiver Queue requires a lock operation. With the proposed mechanism, frequent thread distributions for each virtual processor have little influence on an application program. The reason is that the proposed scheduling mechanism requires no lock operation. Consequently, Lesser Bear always lets each virtual processor have an appropriate number of threads, each virtual processor keeps executing threads, and CPU utilization is improved.

In the proposed mechanism, threads move among virtual processors according to the diagram shown in figure 5. We are concerned about the moving cost from VP_i

to $VP_{backward(i)}$. For example, when running a fork-join type application program, a large number of threads may be created in VP_i. Delivering threads to $VP_{backward(i)}$ may require a lot of time. However in the previous version of Lesser Bear, this process is serialized until each virtual processor has threads because each virtual processor uses a lock operation. The proposed mechanism requires no lock operation, and can distribute threads to each virtual processor in a short time. We confirm this in the next section.

4 Evaluation

In this section, we compare and evaluate the previous version of Lesser Bear (referred to as LB1) and the new version of Lesser Bear that is implemented the proposed scheduling mechanism (referred to as LB2). Experiments are conducted on a Sun Microsystems SPARC Server 1000 running SunOS version 5.5.1. 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. For an application program, we use the radix sort program.

4.1 Radix-sort

The radix-sorting algorithm [21] 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 takes the value of 2 raised to the power n. By distributing all keys, it is easy to execute a radix-sort program in parallel, and we can expect to achieve high scalability.

For example the radix is 4, we separate a set all of the keys for each thread and sort each thread in order to parallelize the radix sort algorithm by thread programming as follows ([8]).

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 data that previous threads create.
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 [22], which is adopted for the interface of Lesser Bear, does not support the barrier synchronization. In this experiment, we implement a barrier synchronization by utilizing creation and termination. Surplus threads are terminated in serial part, and necessary threads are created in parallel part.

4.2 Comparison with the previous version of Lesser Bear

We first compare the time required to distribute threads equally for all virtual processors. The LB2 can distribute threads at a low-cost because no lock operation is necessary for thread movement between peer virtual processors. A fork-join application dynamically changes the number of running threads. Moreover, low-cost distribution prevents a virtual processor from being idle and improves CPU utilization. Therefore, threads should be distributed equally and quickly.

Table 2: Cost of distribution for every virtual processor ($m$sec).

	LB1	LB2
time	131.6	30.8

Table 2 shows the time is spent in changing from serial to parallel inside of the application. In changing from serial to parallel, this application acts as following

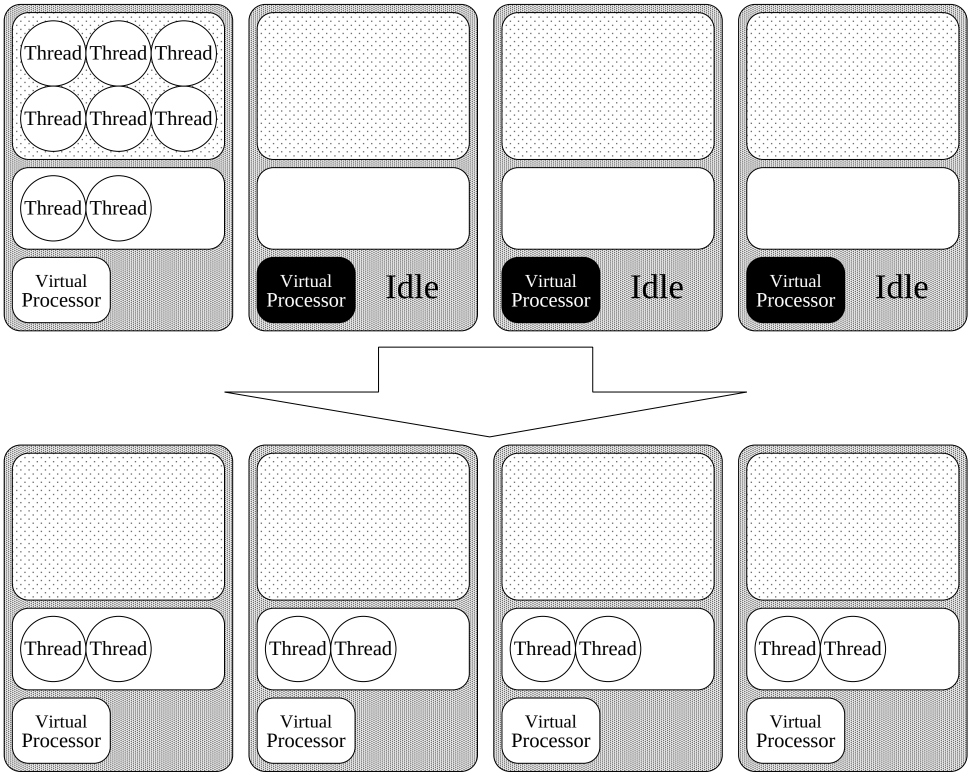

Figure 6: Distribution for maintaining the number of threads equally for all virtual processors.

(figure 6):

- A large number of threads are created at a time.
- They are distributed equally to every virtual processor.

In the LB1, a lock operation is required for the thread movement, but the LB2 requires no lock operation. Table 2 shows the result of this improvement.

We measure the turnaround time of the LB1 and LB2. In the application of this experiment,

- the number of keys is 2^{22},
- 2^{8} threads are created, and

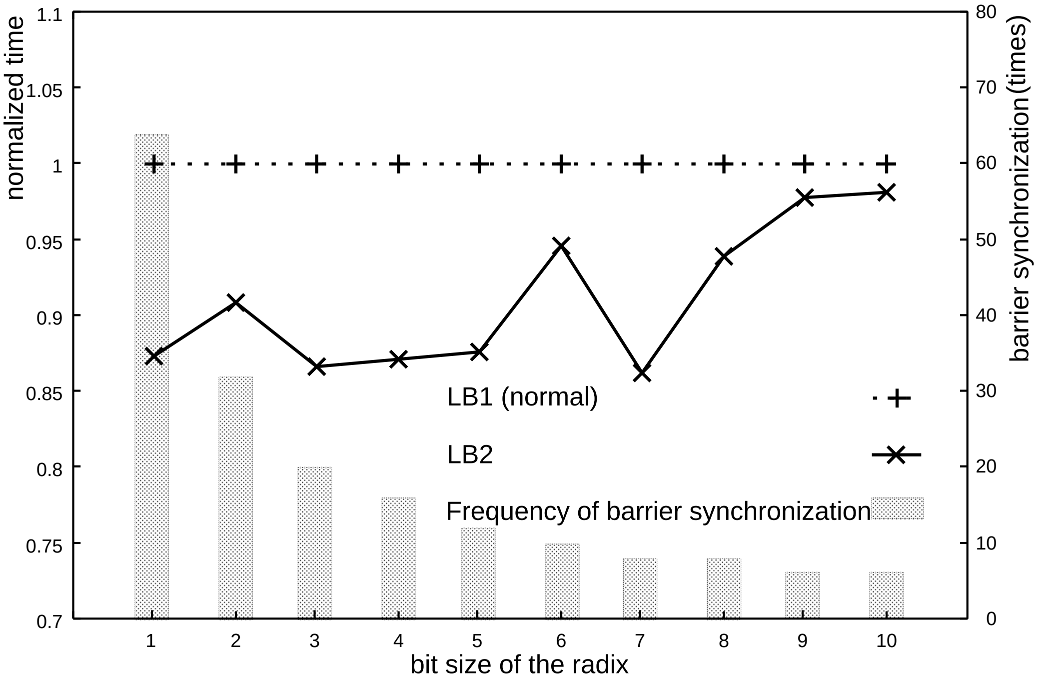

Figure 7: Comparison of turnaround time between the LB1 and the LB2.

- we vary the size of the radix (2^1, 2^2, $\cdots$, 2^{10}), and measure the turnaround time.

In Figure 7, the horizontal axis of the graph represents the size of the radix, and the vertical axis represents execution time normalized to that of the LB1.

The implemented radix sort program has following features.

- The smaller the radix size is, the more a time this program loops inside and the more frequently barrier synchronization occurs.

- Frequent barrier synchronization contributes to system overhead.

When the radix sort program is run, the smaller the radix size is, the better performance the LB2 achieves compared with the LB1. This means that there can be a lot of thread managements even when there is a small radix. The lock operation is not necessary for thread managements in the LB2, so that this mechanism performs well with a small radix. When the radix is 2^1, a barrier synchronization is generated 64 times and the turnaround time is reduced by about 14 %.

Table 3: Idle time for each virtual processor ($m$sec).

VP_0	VP_1	VP_2	VP_3	VP_4	VP_5	VP_6	VP_7
167.9	4211.4	4178.8	4171.6	4439.5	4614.1	4129.0	4292.7

Table 3 shows idle time for each virtual processor. Whenever a virtual processor becomes idle, it falls asleep in the LB2. Therefore, idling time is equal to sleeping time.

All virtual processors but virtual processor 0 are idle for about 4 seconds, because creating the data (2^{22} keys) takes that long. Furthermore, the application does not create any threads while it creates the data. From the result, we can say that virtual processors are hardly idle relatively in LB2. Even if a virtual processor becomes idle, it seems to gain threads in a short time.

4.3 Comparison with Solaris threads

We also compare the LB2 and Solaris threads by executing the radix sort program.

Solaris threads is a thread library supported by SunOS 5.x as described in **2.1**.

Solaris threads requires kernel support for thread managements, so that we can not expect that it performs so well in an application in which thread management occurs.

In this experiment, we use Solaris threads as follows:

- We use the stack by the same method as Lesser Bear. For Solaris threads, we use `pthread_attr_setstackaddr()` and `pthread_attr_setstacksize()` functions.

- We use the suitable number of LWPs by `thr_setconcurrency()` function.

In addition to these, a barrier synchronization is implemented by utilizing mutex variables and condition variables for this experiment [23].

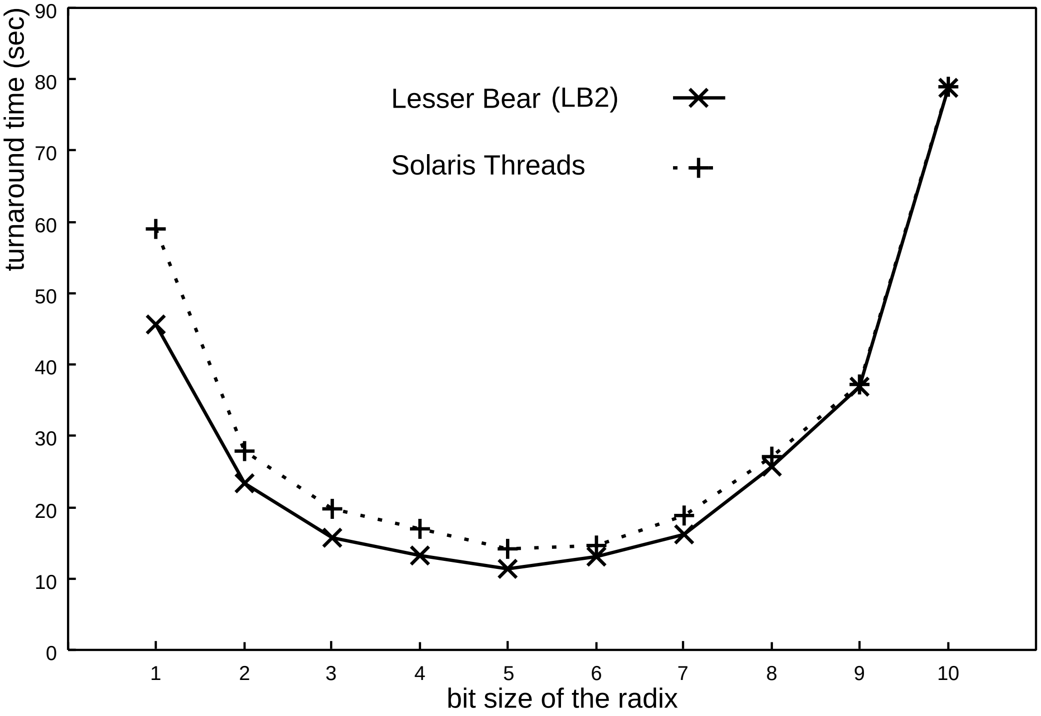

Figure 8: Comparison of turnaround time between the LB2 and Solaris threads.

Figure 8 shows a comparison between the LB2 and Solaris threads. The horizontal axis of the graph represents the radix size. The vertical axis represents the turnaround time.

When the radix size is small, fork-join operations occur frequently, and the number of serial parts increases inside of the application. Figure 8 shows the application features.

Figure 8 shows that the LB2 has good performance when the radix is small. This is mainly because the thread management's overhead of Solaris threads is high.

Lesser Bear has improved mutex variable control and condition variable control by preparing two queues such as the Protect Queue and the Waiver Queue, and this reduces the control overhead [8]. In this experiment, when the radix is 2^1, the turnaround time is reduced by about 23 %.

4.4 Comparison with LinuxThreads

At last, we compare the LB2 and LinuxThreads by executing the radix sort program.

We use an SMP PC system running Linux 2.4.2 for this experiment. This system has four CPUs (500MHz Intel PentiumIII Xeon processors) and 256MB of main memory.

LinuxThreads is integrated in GNU C library, and supplied by almost all the Linux distributions. However, LinuxThreads provides kernel-level threads and scheduling between threads is handled by the kernel scheduler. Therefore, we can expect that the experimental result of LinuxThreads is similar to that of Solaris threads.

In this experiment we use the radix sort program as same as section 4.3.

Figure 9 shows a comparison between the LB2 and LinuxThreads. The horizontal axis of the graph represents the radix size. The vertical axis represents the turnaround time.

This result is similar to the graph shown in section 4.3. This is mainly because the thread managements of LinuxThreads requires kernel faculties. Hence LinuxThreads is not suitable for an appliaction like the radix sort program. In this experiment, when the radix is 2^1, the turnaround time is reduced by about 14 %.

These results show that the LB2 achieves low-overhead thread management and high CPU utilization.

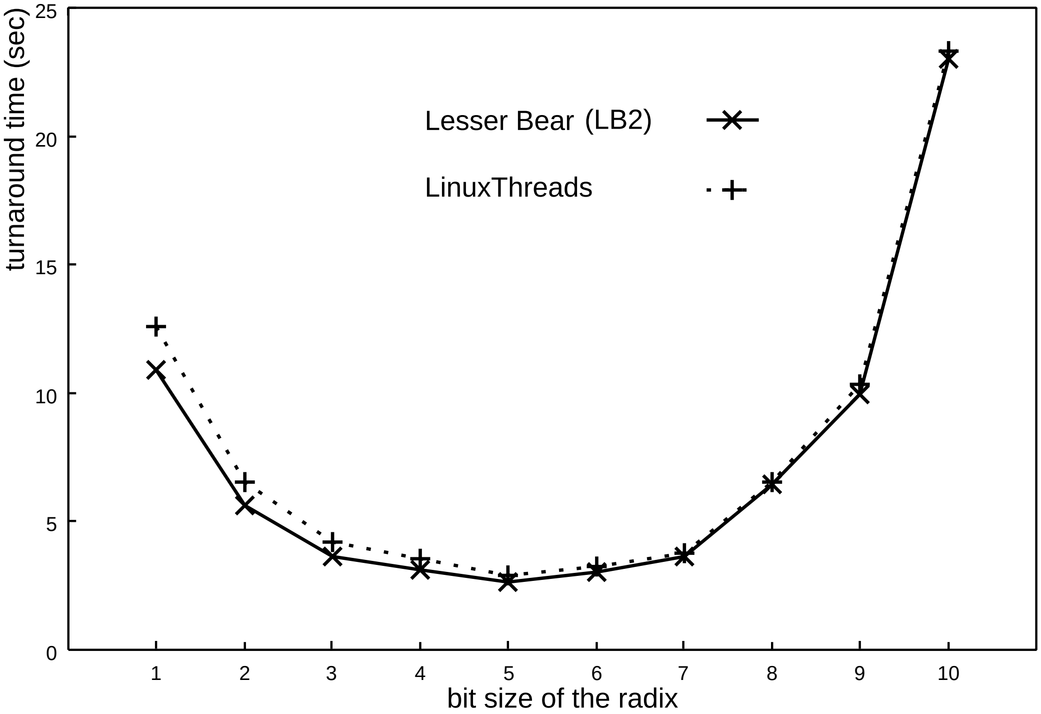

Figure 9: Comparison of turnaround time between the LB2 and LinuxThreads.

5 Conclusions

In this paper, we have proposed a scheduling mechanism that eliminates lock operations and reduces scheduling overhead.

To eliminate lock operations, we utilize the feature of the Protect Queue and the Waiver Queue that are implemented in the previous version of Lesser Bear. First, each partial shared-memory space, which is divided as a working space for each virtual processor, forms link topology through the Waiver Queue. Next, we fix inserting virtual processor and removing virtual processor for each Waiver Queue. For enqueueing and dequeueing operations, we use a lock-free algorithm. We therefore accomplish a rotatory scheduling mechanism requiring no lock operation. This scheduling mechanism enables Lesser Bear to reduce the overhead of moving threads among virtual processors. Moreover, it can keep the amount of threads for each virtual processor distributed equally.

We have used a radix-sort program as the application program in the experiments. We have confirmed the effect on elimination of a lock operation.

From the results of running the application, we have confirmed that the overhead of thread managements in the proposed scheduling mechanism is lower than the overhead of the previous version of Lesser Bear, that of Solaris threads and that of LinuxThreads. We have also confirmed that the proposed scheduling mechanism achieves high CPU utilization.

References

[1] 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.

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

[3] Cattaneo G, Giore GD, Ruotolo M. Another C threads library. *ACM SIGPLAN Notices* 1992; **27**(12):81–90.

[4] Mueller F. A library implementation of POSIX threads under UNIX. *Proceedings of the Winter 1993 USENIX Conference*, San Diego, California, USA. USENIX Association: Berkely, 1993; 29–41.

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

[6] Solaris 2 Migration ToolKit – Online Documents.
http://www.sun.com/smcc/solaris-migration/docs/whitepapers.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. *Concurrency and Computation: Practice and Experience*; to appear.

[9] Suzuki M, Watanabe T. A lightweight solution for the producer-consumer problem, *Report CS 00-05*, Department of Computer Science, University of Electro-Communications, Tokyo, 2000.

[10] 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 and Computation: Practice and Experience* 2001; **13**(2):141–152

[11] Edler J, Lipki J, Schonberg E. Process management for highly parallel UNIX systems. *Proceedings of the USENIX Workshop on UNIX and Supercomputers*, Pittsburgh, Pennsylvania, USA. USENIX Association: Berkely, 1988; 1–18.

[12] Kontothanassis LI, Wisniewski RW, and Schott ML. Scheduler-conscious synchronization. *ACM Transactions on Computer Systems* 1997; **15**(1):3–40.

[13] Krieger O, Stumm M, Unrau R, Hanna J. A fair fast scalable reader-writer lock. *Proceedings of the 1993 International Conference on Parallel Processing*, Williamsburg, Virginia, USA. CRC Press, Boca Raton, 1993; volume II: 201–204.

[14] Mellor-Crummey JM, Scott ML. Algorithm for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems* 1991; **9**(9):21–65.

[15] Michael MM, Scott ML. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprosessors. *Journal of Parallel and Distributed Computing* 1998; **51**(1):1–26.

[16] Prakash S, Lee YH, Johnson T. A nonblocking algorithm for shared queues using compare-and-swap. *IEEE Transactions on Computers* 1994; **43**(5):548–559.

[17] Mukherjee R, Bennett JK. Operating system design principles for scalable shared memory multiprocessors. *Proceedings of the 8th ISCA Conference on Parallel and Distributed Computing Systems*, Orlando, Florida, USA. International Society for Computers and Thier Applications: Raleigh, North Carolina, USA, 1995; 248–255.

[18] Nikolopoulos DS, Polychronopoulos ED, Papatheodorou TS. Efficient runtime thread management for the nano-threads programming model. *Parallel and Distributed Processing, 10 IPPS/SPDP '98 Workshops Held in Conjunction with the 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing*, Orlando, Florida, USA (*Lecture Notes in Computer Science*, vol. 1388). Springer-Verlag, Berlin, 1998; 183–194.

[19] Dandamudi SP, Cheng PSP. A hierarchical task queue organization for shared-memory multiprocessor systems. *IEEE Transactions on Paralleland Distributed systems* 1995; **6**(1):1–16.

[20] Oguma H, Nakayama Y. A scheduling mechanism for lock-free operation of a lightweight process library for SMP computers, *Proceedings of the 8th International Conference on Parallel and Distributed Systems*, KyongJu City, Korea. IEEE Computer Society: Los Alamitas, 2001; 235–242.

[21] Zagha M, Blelloch GE. Radix sort for vector multiprocessors. *Proceedings of the 1991 conference on Supercomputing '91*, Albuquerque, New Mexico, USA. IEEE Computer Society: Los Alamitas, 1991; 712–721.

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

[23] Lewis B. Multithreaded Programming Education. http://www.lambdacs.com/books/books.html.