オペレーティング・システム 51－3
（１９９１．７．１８）

MACHカーネルにおけるマルチプロセッサ同期機構の解析

三菱電機株式会社　情報電子研究所
Joe Uemura,　坂倉隆史、菅隆志

対称型マルチプロセッサでは、OSカーネルの共有資源へのアクセス競合が性能に大きな影響を与える。そこで、この動的特性を知るため、カーネギーメロン大学で開発されたMACHを用い並列処理同期機構の解析を最大４CPUのシステム上で行なった。MACHでは様々なアクセス競合に対し数種類のロック機構を用い細かな粒度での並列実行を行っている。本解析では、特に分散／並列OSで重要となるメッセージ交換とメモリ管理の機能に注目し、数千個以上のロック対象箇所を数十のクラスに分け、アクセス頻度、競合度、獲得時間などを解析した。その結果、競合度は1%程度であることなどが実験的に明らかになった。

An Empirical Investigation of Multiprocessor Synchronization Mechanisms in the MACH Kernel

Joe A. Uemura, Takashi Sakakura & Takashi Kan

Mitsubishi Electric Co.
Computer & Information Systems Laboratory
5-1-1 Ofuna
Kamakura, Kanagawa

Mach is a distributed multiprocessor operating system developed at Carnegie Mellon University. In this paper, we evaluate the multiprocessor synchronization mechanisms used internally in the MACH kernel on a four-cpu multiprocessor. The MACH core kernel provides parallel execution by using locks to encapsulate shared resources in critical regions. We wrote two programs to evaluate how those locks behave. These programs stress message handling and virtual memory management, two key kernel functions for building client-server type of applications. Our examination focused on lock contention and activity issues; our results show that most locks have low contention (around 1%), and there is no substantial waiting time when contention occurs.

1. Introduction

MACH is a distributed multiprocessor operating system developed at Carnegie Mellon University (CMU) [Tevanian87] [Accetta86]. MACH has been widely used as a research vehicle for experimenting with new architectures for multiprocessors and distributed systems. MACH has undergone several revisions, and been distributed in various different forms. Notably, the Open Software Foundation has based OSF/1, their operating system component, on MACH technology, and Carnegie Mellon University has recently released a new version of MACH, version 3.0, entirely based on a microkernel architecture. We have ported MACH Version 2.5 obtained from CMU to an experimental four RISC, shared-memory, symmetric multiprocessor system. The work described in this paper was conducted using this multiprocessor.

Generally speaking, MACH Version 2.5 has two main parts: the MACH core kernel, made of native code which provides new functionality developed at CMU, and the 4.3BSD compatibility code. The MACH core kernel provides only generic primitive services such as virtual memory management, interprocess communication, and process scheduling. This core kernel forms the basis of the microkernel architecture currently being developed at CMU. This kernel was developed from scratch and has been fully parallelized mainly using locks as synchronization mechanisms. This code executes correctly on multiprocessors while providing fine-grained parallel kernel execution.

The UNIX[1] compatibility code is made mostly of 4.3BSD code. As such, this code was not originally designed to run on multiprocessors. MACH resolves this problem by simply forcing this code to be executed on a single processor known as the unix master processor.

This paper first introduces the multiprocessor synchronization mechanisms used in the core kernel, and describes how the unix master mechanism ensures proper execution of the UNIX compatibility code. It then reports on the experiments we conducted to investigate the synchronization mechanisms running behavior. Finally, it concludes by presenting several observations based on the results of the experiments.

[1]Developed and Licensed by UNIX System Laboratories, Inc.

2. Multiprocessor Synchronization Mechanisms in the MACH Kernel

2.1. Multiprocessor Support in the MACH Core kernel

The MACH core kernel is a multithreaded kernel. A multithreaded kernel allows multiple threads to execute concurrently in the operating system. In order to provide this type of parallelism, kernel data structures must be protected from uncontrolled access to ensure that kernel resources are kept in a consistent state. Bach[Bach84] and several others [Sinkewicz88] [Campbell91] have described the need to provide synchronization mechanisms for correct kernel execution in the presence of multiprocessors. MACH main synchronization mechanisms are based on lock variables.

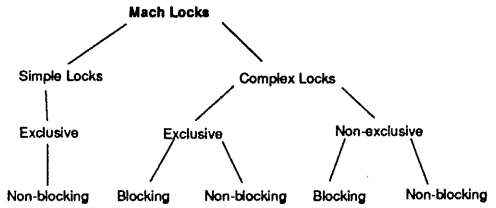
Lock variables are used to implement critical sections. Critical sections restrict access to shared data by providing atomicity of operations performed inside of those sections. Before a thread can enter a critical section and access shared data, it must first acquire the lock protecting the section. This synchronizes threads executing on multiple processors, and ensures that shared data is kept in a consistent state.

MACH implementation of locks is divided into two types: simple locks and complex locks. Simple locks are implemented using a simple spinlock. The spinlock is implemented using a variable which indicates if the lock is idle (unlocked) or busy (locked). If the lock is busy, a thread will spin waiting until the lock becomes idle and can be acquired. All threads trying to acquire a lock will compete for the lock. On shared-memory bus based machines, this requires some form of hardware support to guarantee atomic memory accesses via a memory bus locking protocol.

Simple locks are mutually exclusive locks. Mutually exclusive locks restrict access to a critical section to a single thread. This is true regardless if a thread has intention to modify the data or not. A thread, once having acquired the lock, has exclusive access, and is free to read or write the shared data without any competing access.

Complex locks can, on the other hand, be used to implement both exclusive and non-exclusive locks. Consider the scenario where a thread only wants to read a shared data. As long as all threads concurrently accessing the data are not modifying it, there is no need to have mutually exclusive access - all readers of the data can concurrently read it without causing consistency problems.

Another basic difference between complex locks and simple locks is that complex locks can be both blocking and non-blocking. Blocking locks, unlike spin locks, will cause the thread trying to acquire the lock to block instead of busy waiting until the lock is free.



2.2. Multiprocessor Support in the UNIX Compatibility Code

MACH provides 4.3BSD UNIX binary compatibility by leveraging code as released by UC-Berkeley. Most of this code has been integrated unmodified to the MACH core kernel; only few places required changes. Since this code was not originally designed to run on multiprocessors, some form of multiprocessor control was necessary. The strategy adopted by MACH 2.5 is to limit execution of this code to a single processor - the master processor.

The implementation of this strategy is accomplished by inserting calls to an internal function called ***unix_master*** before executing BSD code. This function checks if the processor trying to execute unparallelized BSD code is already the master processor. If that is not the case, it forces a context switch to the master processor. After the BSD code finishes, a call is made to another internal function, ***unix_release,*** to allow execution to continue on any available processor.

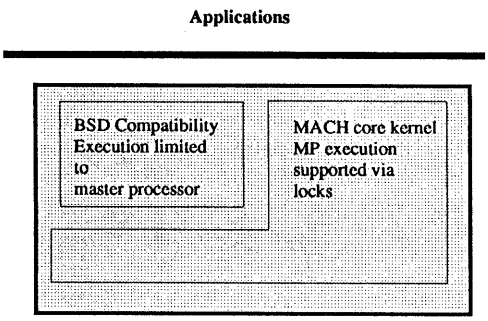


Figure 1.
MACH2.5 Kernel has two main components: the fully parallelized core kernel, and the BSD compatibility code. Multiprocessor execution is supported in the core kernel using locks as synchronization mechanisms whereas execution of BSD code is limited to a single master processor.

It is interesting to note that the issue of parallelizing the compatibility code has been addressed in [Boykin89]. OSF has leveraged this code in their OS component, OSF/1. We look forward to conduct a similar investigation as described in this paper on OSF/1.

3. Investigation of Multiprocessor Synchronization Mechanisms

Our investigation had several motivations. Our interests focused on finding out what these synchronization mechanisms were, and how they were being used. We also wanted to determine their effectiveness and limitations since they should give a good indication of the multiprocessors execution behavior. This was done by examining how stressed the locks were. Finally, we wanted to explore how the locks implementation addressed performance issues.

We decided to focus on the MACH core kernel since it fully supports fine-grained parallel execution. This decision was quite obvious since the compatibility UNIX code in our MACH release is not parallelized. Furthermore, the core kernel forms the basis of the microkernel version of MACH. The restructuring of traditional monolithic operating systems into several servers running as user tasks on top of a microkernel is a recent trend in the operating system research community [Guillemont91] [Golub90]

[Tanenbaum90]. Our examination should give some insight of how the microkernel MACH handles multiprocessors.

As explained above, the core kernel enforces proper multiprocessor execution by protecting critical shared data with lock variables. We decided to concentrate on three main characteristics concerning these locks. We focused our examination on the following issues:

- activity issues
  We wanted to find out how active a lock was during a certain load. This was accomplished by measuring how many times a lock was acquired during a given period of time. This should allow us to estimate the overhead incurred due to locks.

- contention issues
  We wanted to find out how "hot" a lock was by measuring how many times there were contentions when trying to acquire the lock. We measured the ratio of contention vs. lock acquisition.

- latency/granularity issues
  Once there was a contention, we wanted to determine the latency incurred until the lock was finally acquired. This should be basically determined by the granularity, ie. the size, of the area protected by the lock. This measurement was accomplished by monitoring the amount of spins a non-blocking lock spent until acquiring the lock. On a blocking lock, we collected the time elapsed between trying to grab the lock and the actual lock acquisition.

To simplify data gathering and provide a higher view of lock behavior, we collected data on a lock class basis. Instead of gathering information about each instance of a lock in the system, we divided the locks into classes according to their semantic function. For example, all the locks protecting a certain table were put together on a class, and their behavior monitored collectively. Thus, instead of having to deal with a multitude of lock information - in our system, the number of initialized locks at any given time was in the order of several thousands - we had a much more manageable and identifiable set of data. We divided the locks into fifty two classes of simple locks, and nine classes of complex locks. Several of these classes only have a single lock instance, while others have multiple instances within the same class.

To generate lock data, we wrote two simple programs. We were careful to make sure that the programs produced meaningful data about the core kernel. We did not want to write programs which would spend most of the time in the ***unix_master***, and yet we wanted programs that represented a load which would likely to occur in real usage.

The first program we wrote simulated a load typical of client-server style of applications. This should stress the system ability to provide efficient interprocess communication (IPC) on multiprocessors. On a message passing based microkernel architecture such as MACH, the core kernel only provides the basic services such as support for message handling. Since most of the functions available in traditional operating systems run in user-state servers, the core kernel ability to efficiently handle message passing is essential to performance. This message-passing program is made of n clients which issue request to a single server. The server receives each request and replies back to the client issuing the request. Since we wanted to measure kernel load, we made the requests null, meaning that all the server does upon receiving a request is to immediately send a reply to the client. This should stress the kernel's ability to handle a high volume of IPCs typical of this type of application.

The other program was designed to stress virtual memory management, another core kernel functionality. The program has several tasks which share a region set copy-on-write. As long as the tasks only read this region, the kernel does not get involved. However, as soon as a task tries to modify the shared data, a page fault occurs, and the kernel has to allocate a new page for the task to modify. The test program was designed so the tasks will actively modify the shared data, thus stressing the virtual memory subsystem.

3.1. Investigation of Simple Locks

From all fifty two simple lock classes, about twenty to thirty percent encountered contention while running both the message passing program and the VM program. Besides illustrating this relationship,

Figure 2 also shows the number of active[2] classes on both programs.

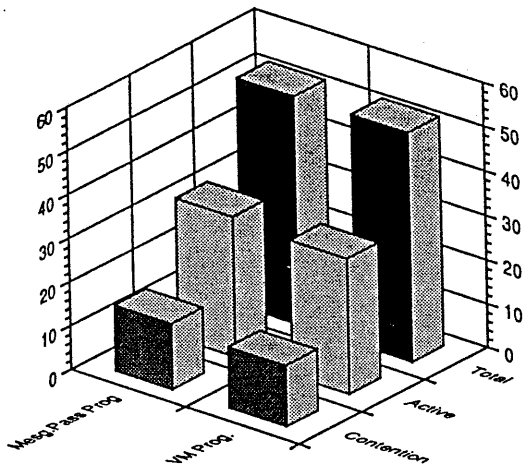


Figure 2

The labels numbered 0~60 represent the number of classes.

Lock Class	ins	grab	confli	co/gr	spin max	spin ave
1. CPU_LIST_CLASS	74	822000	55232	0.067	414	7
2. ZONE_CLASS	33	839412	27019	0.032	613	8
3. OBJ_LOCK_CLASS	445	1849730	15016	0.008	286	11
4. VM_PAGEQ_CLASS	1	360563	10258	0.028	1635	18
5. VM_PAGE_BUCKET	1	300674	7883	0.026	35	5
6. VM_CACHE_CLASS	1	240300	6808	0.028	430	3
7. VM_PAGEQ_FREE_	1	120280	1642	0.014	24	8
8. PMAP_CLASS	73	514778	979	0.002	44	11
9. MESSAGEQ_CLASS	445	320540	381	0.001	220	18
10. THREAD_LOCK_CL	81	562419	209	0.000	14	8
11. GLOBAL_RUNQ_CL	1	138987	172	0.001	23	8
12. VM_OBJECT_CLAS	764	958663	25	0.000	2299	440
13. KERNEL_PMAP_CL	1	137228	18	0.000	39	23
14. QUANTUM_ADJ_CL	1	34	12	0.353	38	21
15. LOCAL_RUNQ_CLA	128	98	1	0.010	10	10
16~52. OTHER CLASS	---	1625552	0	0.000	----	---
TOTAL	---	8791258	125655	0.014	----	---

Table 1.

Table 1 illustrates simple lock behavior running the message passing program on four processors. As can be seen, the contention/grab ratio is fairly low. Those locks which have high contention/grab ratio seem to have low max and average spin times. On the other hand, those with high max & average spin times seems to have low conflict/grab ratios.

Tables 1 & 2 illustrate the behavior of simple locks running each test program respectively on four processors. The data has been sorted by the number of conflicts occurred during the program. Only those locks which experienced conflicts during the test programs are completely depicted.

The table lists the class to which the lock belongs, the number of instances within the class, how many times the lock was grabbed during the program, how many conflicts and the ratio of conflict/grab, and the maximum and average spin counts.

A quick look at this data shows that the overall ratio of conflict to grabs is fairly low[3]. Furthermore, by looking at the maximum and average number of spins, it shows that those locks which have some contention do not appear to have to wait very long to acquire the lock. This is important since the worst scenario would be to have a lock with both high contention/grab ratio and high acquisition latency.

Lock Class	ins	grab	confli	co/gr	spin max	spin ave
1. VM_PAGEQ_CLASS	1	4125	96	0.023	156	29
2. CPU_LIST_CLASS	19	1165	44	0.038	4	2
3. VM_OBJECT_CLAS	127	7366	21	0.003	885	261
4. VM_PAGE_BUCKET	1	3961	15	0.004	949	68
5. OBJ_LOCK_CLASS	153	890	8	0.009	38	19
6. VM_CACHE_CLASS	1	1573	7	0.004	435	101
7. KERNEL_PMAP_CL	1	1327	6	0.005	40	19
8. VM_PAGEQ_FREE_	1	2305	3	0.001	19	8
9. GLOBAL_RUNQ_CL	1	224	1	0.004	12	12
10. PMAP_CLASS	18	1927	1	0.001	54	54
11. ZONE_SPACE_CLA	1	133	1	0.008	8	8
12~52. OTHER CLASS	---	12138	0	0.000	----	---
TOTAL	---	37134	203	0.005	----	---

Table 2.

The VM program runs for a much shorter period than the message passing program.. The number of grabs and conflicts reflect this. However, the ratio conflict/grab and max and average spin have similar characteristics as the numbers in table 1.

To further probe the simple locks behavior, we monitored how the presence of a different number of processors affected the ratio of conflict to grabs. Figure 3 illustrates this

[2]A class was considered active whenever at least one lock within the class was acquired during the monitored program.

[3]By low, we mean <= 1%.

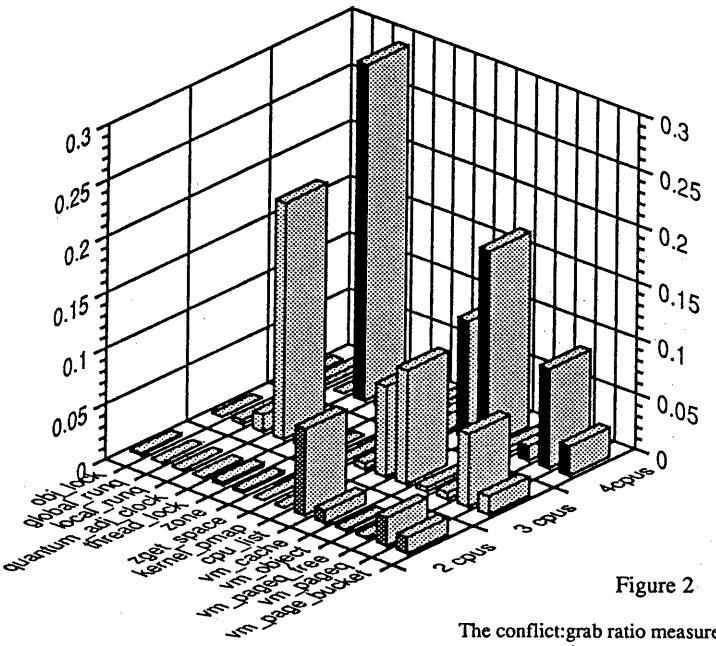


Figure 2

The conflict:grab ratio measured running the message passing program on 2~ 4 processors.

						write conflict					read conflict		
							conc-re		wait			wait	
Lock Class	ins	w-grab	r-grab	sle	writ	read	max	ave	max	ave	conf	max	ave
VM_MAP_CLASS	81	365772	123021	yes	8	5	1	1	8729	4358	3	15532	9283
PMAP_SYSTEM_CL	1	248871	61837	no	7	9266	2	1	13	1	4416	2	1

Table 3

Measurements of complex locks running the message passing program on four processors

						write conflict					read conflict		
							conc-re		wait			wait	
Lock Class	ins	w-grab	r-grab	sle	writ	read	max	ave	max	ave	conf	max	ave
VM_MAP_CLASS	26	4025	1229	yes	16	8	1	1	5204	1543	0	0	0
PMAP_SYSTEM_CL	1	5258	297	no	0	3	1	1	25	9	3	1	1

Table 4

Measurements of complex locks running the VM program on four processors

Data displayed are: lock class name, number of instances in class, number of grabs for read, number of grabs for write, sleep (blocking) lock, write conflicts, and read conflicts. Write conflict data is further broken down by the cause for the conflict: (a) conflict due to another active writer, or (b) conflict because of other reader(s). In case of readers, the numbers show the maximum and average number of concurrent readers. The maximum and average wait times are shown; for blocking locks, these are internal clock units where one unit==250ns.

relationship when running the message passing program. As can be seen, the conflict:grab ratio seems to remain rather low when running the program on two processors. On three processors, there seems to be some more activity. This activity seems to peak at four processors. Fortunetely, even at four processors, most of the locks still show a rather low ratio; only few experience a ratio higher than 0.01, or 1%.

3.2. Investigation of Complex Locks

Tables 3 & 4 illustrate the behavior of complex locks running both tests programs on four processors. As with simple locks, we only display those locks which were active and experienced contention during the test programs. As shown, only two classes of complex locks had conflicts during our test programs. One lock was non-blocking meaning that the lock span when there was a conflict. The other lock was a blocking lock; whenever a thread failed to acquire the lock, it blocked itself until the lock was released.

Both locks were read/write locks. Read/write locks, unlike simple locks, are non-exclusive locks on reads, but exclusive on writes. This means that there can possibly be several threads concurrently reading the data as long as there is not an active writer. This generates extra parallelism by having concurrent non-exclusive access to data. On the other hand, an active writer implies that no reader can be concurrently accessing the data.

This generates several possible scenarios which result in contention when trying to acquire the lock. First of all, the lock can be acquired for read or for write. If a thread is trying to acquire the lock for read, a conflict would occur if there is an active writer, or there is a registered intention to acquire the lock for write (see below). The thread would have to either spin wait (non-blocking lock) or suspend itself (blocking lock) until the writer finishes modifying the data and the lock can be acquired.

If a thread wants to modify the protected data, it has to acquire the lock for write. There are two possible scenarios which would lead to a conflict:

- there is at least one active reader currently accessing the data, or
- there is another active writer currently modifying the data

In the first scenario, MACH, in order to avoid indeterminate wait in case other readers keep arriving and gaining access to the lock, allows the thread to register its intention to acquire the lock for write before waiting/blocking. Any reader arriving after this intention has been declared will wait until the writer gets a chance to modify the data. In the second scenario, the thread will also declare its intention to modify the data, and then block/wait until the current writer finishes.

Tables 3 & 4 display several information collected on complex locks while running the test programs on four processors. As can be seen, the number of grabs for read is quite higher than the number of grabs for write, thus showing the effectiveness of read/write locks. Without read/write locks, all those accesses would be serialized yielding a lower degree of parallelism.

When examining the wait times, we were particularly interested in the number of spins in the non-blocking lock. One of the guidelines for choosing blocking vs. non-blocking locks is the amount of time, or the number of spins, a lock experiences in case of contention. The trade-off is the cost of context switching vs. wasting processor power. If the lock were to spin a large number of times, it would be better to block the thread and let the processor do something more useful than simply spin. However, if the number is not high enough to justify the context switch costs required to dispatch the processor to another thread, spinning becomes a more plausible alternative. The non-blocking pmap_system lock displays low spin times which seems to justify the choice of making this a non-blocking lock.

4. Conclusion

MACH support for multiprocessor is based on using locks to provide fine-grained parallel kernel execution. We have examined the behavior of these synchronization mechanisms. Our investigation focused on lock activity and contention issues. We examined both simple and complex locks. The simple lock examination shows that from all the identified lock classes, about 60% were active - grabbed the

lock at least once - during our test programs. Among the active locks, about 45% experienced contention. The ratio of contention per number of grabs was fairly low - under 1% - for most locks. Another issue considered was how long it took until a lock could finally be acquired whenever there was contention. This wait period also showed to be generally low for most locks. Those locks which did have high wait times fortunately had a low ratio of contention/grab. If both the ratio of contention/grab and the latency had been high, this would imply the presence of a possible software bottleneck yielding a lower degree of parallel kernel execution.

The complex lock observation demonstrated that there was a 3:1 to 4:1 rate of reads to writes. This high rate suggests that using read/write locks result in a higher degree of parallel execution due to non-exclusive concurrent access to shared data on reads. Also, one lock was a complex non-blocking lock. Our measurements confirmed that this lock displays low wait times thus not justifying the cost involved to block the thread in case of contention.

5. Acknowledgments

Masato Takahashi has provided valuable input on how to conduct our experiments. Porting MACH to our prototype multiprocessor was, of course, a team effort. We would like to thank all those who participated in the porting.

6. References

[Accetta86] M. Accetta, R. Baron, D. Gollub, R. Rashid, A. Tevanian, M. Young. MACH: A New Kernel Foundation for UNIX Development.. *Proc. Summer 1986 Usenix Technical Conference*, 1986.

[Bach84] M.J. Bach, S.J. Buroff. Multiprocessor UNIX Systems. *AT&T Bell Laboratories Technical Journal*, Vol.63-8, Part 2, 1984.

[Boykin89] J.Boykin, A. Langerman. The Parallelization of Mach/4.3 BSD: Design Philosophy and Performance Analysis. *Proc. Distribute & Multiprocessor Systems Workshop 1989*, 1989.

[Campbell91] M. Campbell, R. Barton, J. Browing, D. Cervenka, B. Curry, T. Davis, T. Edmonds, R. Holt, J. Slice, T. Smith, R. Wescott. The Parallelization of UNIX System V Release 4.0. *Proc. Winter 91 Usenix Technical Conference*, 1991.

[Golub90] D.Golub, R. Dean, A. Forin, R. Rashid. UNIX as an Application Program. *Proc. Summer 1990 Usenix Technical Conference*, 1990.

[Guillemont91] M. Guillemont, J. Lipkis, D. Orr, M. Rozier. A Second Generation Micro-Kernel Based UNIX; Lessons in Performance and Compatibility. *Proc. Winter 91 Usenix Technical Conference*, 1991.

[Sinkewicz88] U. Sinkewicz. A Strategy for SMP Ultrix. *Proc. Summer 1988 Usenix Technical Conference*, 1988.

[Tanenbaum90] A. S. Tanenbaum, R. Renesse, H. Staveren, G. Sharp, S. Mullender, J. Jansen, G. Rossum. Experience with the Amoeba Distributed Operating System. *Communications of the ACM*, Dec.1990.

[Tevanian87] A. Tevanian, R. Rashid, MACH: A Basis for Future UNIX Development, *Technical Report (CMU-CS-87-139)*, Carnegie Mellon University, 1987.