Evaluating Locking Schemes in the FreeBSD Kernel

james francis toy iv
Advisor: David Hemmendinger

20100321

Abstract

This project is concerned with locking strategies in the FreeBSD unix kernel. Locks prevent race conditions in a multi-programming environment. At present the FreeBSD kernel employs a GIANT_LOCK to prevent race conditions by locking access to the entire kernel. This method may inhibit concurrency; however, it is a simple and easy way to ensure correctness. The goal of this project is to see if fine-grained locking (FGL), an alternative to the GIANT_LOCK, promotes concurrency and throughput by locking only critical shared resources. FGL was implemented for two kernel subsystems, and we measured the time that a thread was waiting to acquire a lock; FGL showed zero wait time and the GIANT_LOCK showed non-zero wait time.

1 An Introduction to Locking

Locking and synchronization were developed to prevent processes from interfering with one another when having access to shared resources. This is important because in a multi-threaded environment these situations often arise since it is more efficient to execute programs in parallel. This contention for shared data is more commonly described as a "race condition" where multiple processes are racing to a shared resource. These race conditions can be resolved by mutual exclusion (ensures only one process is using a shared resource at a time) which is behavior provided by the locking mechanisms currently employed by the FreeBSD kernel. The purpose of this project is to analyze the current locking scheme in the FreeBSD kernel; attempt to implement a more efficient

locking method, and compare the initial method versus the proposed method in this project.

While fine grained locking can improve the performance of a multi-programming kernel system by allowing multiple threads in kernel space, it also complicates existing kernel code. The reason for this is that in a GIANT_LOCKed environment the kernel employs a singular GIANT_LOCK that is responsible for ensuring only one thread is in the kernel at any one time. All other threads must wait for the kernel resident thread to complete, rescind its heightened privileges, and return to user-space (everything that exists out of kernel space). Considering the facilities that the kernel provides to an operating system, it is important to ensure the greatest progress while threads are in kernel space. Kernel progress may benefit from removing the GIANT_LOCK in favor of subsystem specific locks. Subsystem specific locks means that for every subsystem there are specific locks employed to protect shared, system critical, resources. As a brief example, consider printing to the console, a task which requires kernel resident facilities. Suppose two threads call a kernel print function whose job it is to output strings to the kernel console and log – in this case: "bad" and "dog". In the GIANT_LOCKed environment the system would allow one thread into the kernel, whichever arrived first, and lock all other resources. Assume that the thread printing "bad" is forked first. The console will print the expected "bad"..."dog". If fine grained locking is implemented, both threads are allowed into the kernel. If improper fine grained locking is employed all sorts of unexpected behavior is entirely possible – namely console output that appears : "bda"..."dog" where the first 'd' represents the beginning of dog and the second 'd' represents the ending of the string "bad". This **is** unintended behavior and if the systems calling for these return strings are non-human, they, too, will exhibit unintended behavior upon the grounds that they were given improper

input. Note that the system breaks down in the kernel and proliferates from the kernel throughout the rest of the system. In this rudimentary example, it is easily argued that it is not worth implementing fine grained locking in this specific kernel faculty because the progress that the system stands to gain is not great enough to warrant the code complexity that fine grained locking requires.

1.1 Memory Concerns

In addition to unexpected faulty behavior, memory concerns arise since allowing several threads in the kernel also means wiring memory to these kernel resident threads which draws resources away from the programs that the end user is actually **interacting** with. This is a concern because the kernel is designed to be a facilitator, a means to an end, and not an end itself. If the locking is improperly implemented the operating system and the kernel can become a resource monster and is no longer facilitating so much as utilizing resources that the kernel itself has unquestioned power over. This is not the intended allocation of hardware resources.

1.2 What Matters

The goal of this project is to provide a metric for a couple of things. One of these things is to understand whether removing GIANT, and subsequently complicating the code, is warranted and yields good results. The other is ensuring that if fine grained locking is chosen, ensures that the priority of hardware and system resources are still properly assigned to programs that the user is interacting with. Handling these cases, the systems programmer can be sure that employing a new, more contemporary, locking scheme is justified and is facilitating rather than overloading the system.

2 Background

Three people have made major contributions to concurrency. These men are: Edsger Dijkstra, Tony Hoare, and Per Brinch Hansen. They have provided a fundamental basis for everything that relates to concurrency and parallelism today. It is important to note that just because this project pertains to an operating system, it does not mean that these concepts are not applicable elsewhere.

Today it is hard to fathom how difficult work was in the early days of programming. The programmer was responsible for many more things than getting the code to "just work". The programmer was responsible for things like resource management, CPU time slices, and many other things that are taken for granted by programmers today. In fact, bugs seldom were in user programs; it was usually in the bootstrapping programs that ran the hardware. The hardware bootstrapping difficulties and resource management lead to the inventions of systems that ran on top of the hardware all the time. These large, hardware and resource governing, programs were called: operating systems. Shortly after the idea of an operating system came the notion of multiprogramming. If a program is running the hardware and the programmer can write an application to run on top of that without having to hassle with setting up memory and the like, why can't the programmer run several of these user-land programs at once?! This was an impossibility at the time (unless interleaving computation and I/O) due to the fact that all environments were uni-programming environments that ran only one application at a time, due to the lack of systems software. Often the computer memory was used initially to hold the requisite systems software and later the user program code, data, etc. Paging and CPU scheduling did not exists yet so memory mapping was always one-to-one and often hard-coded. These are only a few of the problems that were present in

early operating systems software.

Once a few of these problems were understood, solutions to these problems were sought vigorously. Edsger Dijkstra, was responsible for the notion of "Co-operating Sequential Processes". This was a pivotal moment in the history of computing; several important aspects of concurrency arose from this paper. Dijkstra explains his "layered approach" [1] to the operating systems concept in great detail. It is very interesting to note his concept of a "critical section" [1], which is a block of code that processes must execute with mutual exclusion. The concept of mutual exclusion is prevalent in today's operating systems; it is also used in applications that execute in either multiple threads processes or general concurrency. Problems similar to the "bad"..."dog" example described above led to the invention of semaphores.

2.1 Semaphores

A semaphore is a primitive that executes atomically. It is important to understand the notion of atomicity in order to understand semaphores. When something is run atomically it is never interrupted. This is essential to the concept of semaphores because when dealing with multiple processes presumably they are executing in parallel. It essentially emulates the act of turning CPU interrupts off for a given thread or set of threads that are synchronized by a semaphore. A final note about semaphores is that there is hardware help involved in their implementation, namely, atomic machine instructions such as test-and-set. Test and set works very simply, it takes a memory location and a register, and sets the memory to one and returns its old value in the register. This is how the binary semaphore was created; it's either locked or unlocked. Now consider a situation where several resources exist and are either available (unlocked) or unavailable (locked). Similar to many other applications in

computer science complex things are composed of the smaller, more fundamental, things. Counting semaphores can be implemented via binary semaphores. Counting semaphores are used for resource management. If there are three resources and two are in use, the counting semaphore is initialized to three and in the case of two resources being in use is set to one (since that is the number of free resources). This is how counting semaphores keep track of resources.

2.2 Solutions to Synchronization Problems

Semaphores are a fundamental construct and basic synchronization tool used to lock operating system code. Semaphores are low-level and fast by test and set hardware help. Brinch Hansen, Hoare, et al. introduced higher-level constructs (such as monitors) in programming languages in order to bring concurrency to programming as a behavior instead of bare synchronization. Within fifteen years (the early 1960's to the mid 1970's) the concept of concurrency had become well-understood. From this point forward operating systems became truly multiprogramming environments. This was **THE** (The Structure of the THE Multiprogramming System 1968 - [3]) proof of concept that the computing society was waiting for. Dijkstra's semaphores played an instrumental role in this process and was the means to the end. Higher-level synchronization constructs simply took semaphores and made them more readable and easier to use. High level locking constructs are an interface to the underlying semaphores that utilize the swift hardware help actions. The computing community is indeed in debt to these men for the amount of thought and pioneering that went into this process. As originally stated it was surprising how little was known in the computing environment then, and for these men to flesh things out in this amount of detail and speed is an amazing feat. Wrapping full circle back to the implementation of this project, I personally will be removing GIANT_LOCKs

and promoting concurrency in a few places with several locks employed in the FreeBSD kernel.

3 Design

3.1 Synchronization Primitives Employed in FreeBSD

At present, FreeBSD employs twelve kinds of locks in kernel space which can be used to replace the GIANT_LOCK with fine grained locks. The "shared / exclusive lock" (sx lock) plays an instrumental role in the implementation of this project (along with a few others). The sx lock can share a critical section with several readers; however, if a thread enters preparing to write it picks up the exclusive component of the sx lock. These features make the sx lock very useful when dealing with subsystems that are seldom changed and often read.

3.2 Method of Evaluation

The most important part of this project is the method of evaluation. In order to properly analyze locking methods we must design and implement an understandable method for comparing the GIANT_LOCKed subsystems to fine grain locked subsystems. As a result, the proposed method is as follows: pick a few subsystems to compare, branch the code repository into two branches (GIANT on one and FGL on the other), simplify the BSD locking profiler output in order to create a (more) human readable result. This evaluation should show when to use fine-grained locking. Any subsystem currently employing a GIANT_LOCK **can** be reprogrammed in a fine grained locking scheme which uses several of the twelve locks that FreeBSD already employs; but ensuring the kernel is doing its job well is the most important part of this reimplementation. Kernel progress means nothing when user progress grinds to a halt. The two main methods of

evaluation that will be used are as follows:

1. Specifically coded test programs (hacks) that will cause a specific behavior in the kernel. These hacks will attempt to break down the newly implemented fine grained locking schemes by trying to produce plausible inefficiencies (e.g. allowing too much memory allocation). This will require thorough testing.

2. Modifying FreeBSD's lock profiler. The profiler provides information about where code is getting stuck during kernel intensive processes. It provides a gamut of useful information such as total wait time, average wait time, and the exact line numbers and file names where threads are waiting at kernel locks.

Properly using these resources will greatly improve the output of the project. This evaluation will show where complicating the code is not worth it; and where using the GIANT_LOCK, with regards to a specific subsystem, is a more practical and clean approach. This will prove useful in operating systems research and implementation because understanding when a change is worth it is half of the battle; specifically when it comes to something as complex as concurrency control. Additionally, it will also provide a platform for understanding when and where a systems programmer must be careful about allocating too much memory to kernel resources; which, in turn, can have adverse effects on the programs that the user actually interacts with.

3.3 Sysctl

Sysctl was an initial target of this project because it is relatively high traffic and is simple to understand (a big getter and setter for the kernel). Consider a linked list structure that holds all of the systems kernel parameters. This linked list is accessed via the sysctl user program. For example, executing `sysctl -a` prints all of the kernel parameters currently set, and NIL for those that remain unset. Concurrency control of this list is paramount since it contains extremely sensitive information. If one thread is run to set a parameter "A" and another thread is forked from a process whose task is to query "A" and return the value, the system must ensure that the threads are not setting and getting the kernel parameter "A" at the same time. A shared exclusive lock is employed to ensure this does not happen. In this example using an sx lock is elegant because it provides the fine grained locking functionality without making the code too complex. Alternatively, sysctl **could** have been programmed to figure out if an incoming thread was reading or writing to the kernel parameter and then employed either a reader's only lock, which would ensure no writers were presently writing, and in the event of no writer, the reading thread would be permitted to read. If however, a writer were to arrive and twenty or so threads were reading this popular kernel variable "A" the writer would be required to wait until all of the readers finished, in order to update the parameter; this could lead to several reading threads returning a stale version of "A" to the requesting programs or users. In addition to the concern about the stale variable this locking scheme makes the code considerably more complex; because now instead of just having a thread try to access a shared resource the kernel programmer would have to implement several tests to decipher what kind of thread is incoming. This is the sort of bad design and complexity that must be avoided when implementing system-critical components.

While there should be greater throughput due to the use of a fine grained locking scheme, it is also important to take into account the other constraint when using fine grained locking : memory management. It is good to allow multiple threads access to kernel parameters through sysctl; however, if twenty threads enter the kernel to read parameter{s} from the sysctl data structures and several, perhaps approximately half, require more than a page of memory to be allocated (or wired) to the thread in order to execute – this becomes a resource intensive memory task. Minimizing the amount of memory allocated to the kernel is important because it ensures the memory is being allocated to user programs instead of favoring kernel tasks. If the user experiences significant slowdown, then fine-grained locking is not helpful. To be sure that memory resources are properly allocated, a simple test prior to the sysctl code allowing a thread to wire memory is to say: if the thread requires more than the kernel's defined page size to be wired, we do not allow the process in until the requests are under the page size. Again, this is an important aspect of locking implementation because it helps to ensure the user is getting the performance in userspace as opposed to the resources being allocated and **remaining** in kernel space. The allocation is resources matters because the ultimate goal is to enable the user, not just the kernel, to wait less. If the kernel becomes a resource hog and does not facilitate more than it does burden the system, it is no longer useful to the system.

3.4 Klog

Klog is responsible for printing and recording kernel events to stdout and syslog respectively. This subsystem is a good choice for several reasons. It is perhaps the simplest subsystem in the kernel; it provides the kernel with a "printf" equivalent. Also, it is a low priority subsystem; this means that it is highly

likely that GIANT_LOCKing klog and loading it heavily will inhibit kernel progress by keeping other more important threads out of the kernel (e.g. a scheduler bound thread). The simplicity of the subsystem coupled with its low priority nature make it a good test case for this experiment.

3.5 Keeping the Hardware Busy

Systems programming is an important aspect of computer science. Unfortunately, due to the lack of interest in this field, systems software is still seriously lacking compared to the system hardware the operating system actually runs on. The operating systems are still unable to fully utilize multiple CPU's in a truly multithreaded environment in which one process can run on multiple CPU's at the same time instead of using only one. This project aims to be a step in promoting better use of system resources in addition to ensuring kernel design is held intact while promoting overall system throughput and progress. The notion of "Keep It Simple" has served systems programmers in innovating elegant ways to solve systems problems. This project, upon evaluation of the selected implementations, will show when it **is** and **is not** worth complicating code for the benefit of kernel progress.

4 Implementation

The BSD kernel is a very complex system. The criteria for choosing the subsystems consist of having a simple FGL implementation and ensuring a high and low traffic representation. The simple FGL implementation will make it easier to be certain the implementation is correct. The high and low traffic dichotomy will provide a metric for measuring the performance of FGL versus GIANT_LOCKing in the extreme cases. Klog and sysctl are the two kernel

subsystems being worked on. Keeping track of locking activity with FGL will be difficult; however, thanks to the locking profiler the job is partially done.

The locking profiler is essentially a character device, much like a driver, that is loaded into the kernel to track all of the locking mechanisms in the kernel (more importantly it tracks the amount of time a thread waits to acquire a lock). There are a non-trivial number of locks in the BSD kernel. As a result of this large number of locks the profiler output, while thorough, is difficult to comprehend without study. This component of the project requires designing a method to: parse the locking data, extract the applicable parts, and present the acquired information in a human readable way. If the experiment yields significant results, it must also have a way of expressing those results to non-specialists.

4.1 Kernel Log (klog)

The klog test requires us to create an unrealistically large number of klog calls. We will initially make it an unrealistic number of calls because we expect this is the most effective way to obtain a measurable difference in locking methods. This test can be done by creating a simple character device that resides in kernel space; this pseudo-driver has only one purpose: to make calls to klog so it can print and record the kernel events. It is also useful to have a user-land program that activates and provide parameters to the pseudo-driver.

An effective parameter can describe the number of kernel events to create during the test. For example calling the program "klogtest –n 100" from the user space would then interact with the pseudo driver and make 100 calls to klog printing digits 1-100. This ensures we are making calls to the subsystem in question.

The purpose of this experiment is not to track the progress of klog itself,

but rather the other, more important, subsystems such as the scheduler and virtual memory. The experimental prediction is that if klog acquires the GIANT_LOCK when it needs to print digits 1-100 it is inhibiting the other (more important) threads from doing what they need to be doing. According to the experimental hypothesis, klog holding the GIANT_LOCK will inhibit kernel space progress.

The two components of the test are: invoking klog to create an control environment (by controlling the number of calls) and analyzing kernel progress via the locking profiler in both the GIANT_LOCKed and FGL implementations. Kernel progress, in this case, is measured in how much time a process is waiting on a lock. Once a control environment has been created, tests to compare FGL to GIANT_LOCK more closely can be conducted. The number of klog calls will be the only variable in this comparison. Klog calls will be decreased until results show the smallest discrepancy between FGL and GIANT_LOCK implementations. Once this minimum value has been determined we can decide whether or not it is enough to make FGL worth implementing in production.

4.2 Sysctl

Many user-land applications use sysctl to get and set kernel parameters. We can utilize the common program "make" in a controlled fashion to procure results. Make provides two very important faculties for this experiment: it calls sysctl for kernel information and it can be run in a multi-threaded manner. These two components make it an ideal way of testing sysctl since we know that this is used in production all the time. The method is as follows: turn on the locking profiler, make the BSD kernel itself with 8 or more threads, turn off the locking profiler and collect elapsed time. The experiment will then repeat for the fine grained locking. In a similar fashion to the klog example, keeping track of the

number of threads will be important in determining at what point, if any, fine grained locking is better than the GIANT_LOCKed scheme. Sysctl is an area of kernel space where reconsidering a locking scheme is applicable because of its high traffic and the service it provides to the user-land.

4.3 Repository Status

At present the code has been branched as described in the project design proposal. The GIANT_LOCKed code remains on the default branch while the fine-grained locked implementation exists on a "feature" branch. Patches to implement fine-grained locking (from their original GIANT_LOCKed state) are available for both klog and sysctl.

4.3.1 KLOG Patch

The patch implementing FGL in the kernel logger is a very simple one. It removes the character device parameter "D_NEEDSGIANT", which is a flag in the required device "cdevsw" struct. The only other thing required in this implementation on a generic level is to lock the character array assigned as the message buffer. This is the only critical section in the code that could be tampered with by another thread.

4.3.2 SYSCTL Patch

The sysctl patch deprecated the use of GIANT in the same fashion. We start by removing the D_NEEDSGIANT flag. The sysctl system requires more design than the klog because it has more functionality. For example sysctl is used for getting and setting kernel space parameters. This means that we have a

reader and writer scenario. The basic idea is if we are reading and another reader comes into the sysctl data structure; let that reader read. Otherwise, the incoming thread is a writer and we must let readers clear before writing. Additionally since the sysctl list contains heterogeneous structures (some ints, floats, char*, etc) there is a possibility of running into it wiring a lot of memory in the event we have a storm of sysctl reading threads inbound. In order to mitigate this, we create a memory constraint on inbound threads of one page. The specific tests designed to show results between GIANT and FGL in the specific implementations are below.

5 Results

The purpose of this project is to evaluate the current locking scheme in FreeBSD; see if the locking methods can be improved, evaluate the changes the new locking scheme introduces and form conclusions based on these changes. Correctness is the most important component of this project. Locking and synchronization are an answer to race condition concerns. Correctness concerns arise when writing to a shared data structure is involved since there may be race conditions. The reason for this is the fact that data read by a reader can be corrupted if a writer does not write atomically. A failing write can lead to stale data, or worse, invalid or corrupt data. In response to these concerns this project makes a few rules about the kind of locks used and the way they protect the shared data structures in the target subsystems. In the case of sysctl one "shared-exclusive" lock is used to protect the linked list data structure comprising the kernel parameters, this lock has two components to it: a shared component and an exclusive component. Readers acquire a shared lock and any number of readers can run in parallel. Writers acquire an exclusive lock, in order to

preserve atomicity, write and then exit to allow the readers access. Using FGLs in this way ensures correctness. In the context of sysctl specifically, readers are what we are most concerned about. This fact appears to conflict with our write then read policy; however, while reading is more common, reading the most up-to-date value is more important. The operating system is concerned with getting kernel parameters in order to make decisions about what to do, for example what architecture the OS is running on. Fortunately most of the parameter setting is done in the boot process so values are seldom updated in a way that inhibits the readers progress.

Table 1: Locking Profiler Output for FGL Implementation

time & type	max held	max wait	total held	total wait	times acquired	average held	average waiting	lock name and location
2010-02-10_12-33.FGL:	163	1592	75194	93989	36551	2	2	kern_sysctl.c:1417 (sleep mutex:Giant)
2010-02-10_12-33.FGL:	1602	0	1702668	0	302728	5	0	kern_sysctl.c:1521 (sx:sysctl lock)
2010-02-11_16-00.FGL:	34	944	45882	10325	33061	1	0	kern_sysctl.c:1417 (sleep mutex:Giant)
2010-02-11_16-00.FGL:	951	0	1101006	0	280373	3	0	kern_sysctl.c:1521 (sx:sysctl lock)
2010-02-13_04-00.FGL:	43	1081	46366	16697	32107	1	0	kern_sysctl.c:1417 (sleep mutex:Giant)
2010-02-13_04-00.FGL:	15	0	56	0	4	14	0	kern_sysctl.c:1513 (sx:sysctl mem)
2010-02-13_04-00.FGL:	1088	0	1078963	0	272328	3	0	kern_sysctl.c:1521 (sx:sysctl lock)
2010-02-13_14-37.FGL:	44	1574	44338	10330	32178	1	0	kern_sysctl.c:1417 (sleep mutex:Giant)
2010-02-13_14-37.FGL:	1581	0	1036385	0	272618	3	0	kern_sysctl.c:1521 (sx:sysctl lock)
2010-02-17_13-40.FGL:	3484	0	6723	0	2	3361	0	kern_sysctl.c:1513 (sx:sysctl mem)
2010-02-17_13-40.FGL:	2093	30486	533695	1046088	192953	2	5	kern_sysctl.c:1417 (sleep mutex:Giant)
2010-02-17_13-40.FGL:	79379	0	13091058	0	1683295	7	0	kern_sysctl.c:1521 (sx:sysctl lock)
2010-02-21_13-53.FGL:	243	12708	65906	132398	31732	2	4	kern_sysctl.c:1417 (sleep mutex:Giant)
2010-02-21_13-53.FGL:	12714	0	1592327	0	273310	5	0	kern_sysctl.c:1521 (sx:sysctl lock)
2010-02-21_13-56.FGL:	16321	0	174581	0	17	10269	0	kern_sysctl.c:1513 (sx:sysctl mem)
2010-02-21_13-56.FGL:	5395	59876	542207	1565156	192675	2	8	kern_sysctl.c:1417 (sleep mutex:Giant)
2010-02-21_13-56.FGL:	59887	0	13842033	0	1686407	8	0	kern_sysctl.c:1521 (sx:sysctl lock)
2010-02-21_14-59.FGL:	379	3801	69063	85525	33169	2	2	kern_sysctl.c:1417 (sleep mutex:Giant)
2010-02-21_14-59.FGL:	14391	0	1599878	0	281596	5	0	kern_sysctl.c:1521 (sx:sysctl lock)

Table 2: Locking Profiler Output for GIANT_LOCK Implementation

time & type	max held	max wait	total held	total wait	times acquired	average held	average waiting	lock name and location
2010-02-21_16-59.GIANT:	399	3599	68010	83615	32179	2	2	kern_sysctl.c:1415 (sleep mutex:Giant)
2010-02-21_16-59.GIANT:	4106	4095	1535226	32843	274957	5	0	kern_sysctl.c:1509 (sx:sysctl lock)
2010-02-21_17-27.GIANT:	852	7709	69253	106485	32409	2	3	kern_sysctl.c:1415 (sleep mutex:Giant)
2010-02-21_17-27.GIANT:	25537	21858	1625217	123366	277537	5	0	kern_sysctl.c:1509 (sx:sysctl lock)
2010-02-21_17-51.GIANT:	417	10870	66444	94647	31925	2	2	kern_sysctl.c:1415 (sleep mutex:Giant)
2010-02-21_17-51.GIANT:	10882	10177	1547534	37990	274321	5	0	kern_sysctl.c:1509 (sx:sysctl lock)
2010-02-21_18-14.GIANT:	183	2011	67471	85530	32376	2	2	kern_sysctl.c:1415 (sleep mutex:Giant)
2010-02-21_18-14.GIANT:	18209	1663	1581871	23312	277656	5	0	kern_sysctl.c:1509 (sx:sysctl lock)
2010-02-21_19-02.GIANT:	3411	89396	566192	1419675	198576	2	7	kern_sysctl.c:1415 (sleep mutex:Giant)
2010-02-21_19-02.GIANT:	89405	99659	14055536	3749684	1699121	8	2	kern_sysctl.c:1509 (sx:sysctl lock)

Table 3: Locking Profiler Output for Improper FGL Implementation

time & type	max held	max wait	total held	total wait	times acquired	average held	average waiting	lock name and location
2010-02-21_16-03.XLOCK:	41867	50474	1612397	177618	2775445	2	2	kern_sysctl.c:1521 (sx:sysctl lock)
2010-02-21_16-03.XLOCK:	259	41857	67805	138275	323712	4	0	kern_sysctl.c:1417 (sleep mutex:Giant)

5.1 Discussion of Lock Profiler Output

The above tables describe the results, collected by the FreeBSD locking profiler, for this project. The first two tables show FGL and GIANT_LOCKed environments while the last table shows the tangential experiment of what happens when FGL is improperly implemented.

The most important part of the profiler output is the "total wait" column because it shows how long a lock (in the lock name and location line) was waiting to be acquired over the life of the test. If this project stands to measure throughput and efficiency, we define the best case scenario as the situation where a CPU is never waiting on a lock acquisition. Focusing on the sysctl lock in the first and second tables, the FGL implementation has zero wait times (across all test iterations) while the GIANT_LOCKed environment has non-zero wait times. The FGL implementation of the sysctl lock **never** waits. Revisiting a component of the project design, sysctl is a kernel construct and data structure that is often read and seldom written to (unless booting), the FGL implementation caters to this fact letting readers acquire a shared lock and writers acquire an exclusive lock which is the reason the results are so good. The max held column is also notable because in the FGL implementation on average the max hold time is smaller; however, the average hold time is very similar. In fact, the average hold time in the GIANT_LOCKed tests is basically constant in both the "Giant mutex" and the "sysctl lock". The reason for this is because the GIANT_LOCKed environment does not offer much flexibility and so consistent time intervals arise (a GIANT_LOCK weakness). Lock acquisitions, when comparing tables one and two, were also consistent with each other which is important because it shows the solution to the waiting problem is not based on the number of times the lock is acquire, but **how** it is acquired.

The third table shows what happens when a fine-grained locking scheme is

improperly implemented. In this case the readers and writers from the sysctl example acquired an "exclusive" lock. This means that any time **any thread** entered the kernel planning to use the sysctl data structures it would need to be the only thread working with the shared resource. At first glance this does not seem to be any different than the GIANT_LOCKed environment; however, the GIANT_LOCK is acquired **once** when entering the sysctl code. The FGL implementation was designed to control the flow of sysctl by distinguishing readers from writers and locking the shared resources accordingly. As a result, there are many places in the sysctl code where a lock can potentially be acquired. In this case that lock is **always** acquired. Looking at the results from the profiler, the acquisition count is astonishing. The "sysctl lock" is acquired two orders of magnitude more than it was in either the FGL or GIANT tests. Additionally the "total wait" time, when compared to GIANT, is approximately five times larger. This shows why implementing FGL must be a careful and properly designed procedure.

5.2 Ensuring Control While Testing

The experiment is governed by a program designed to push the computer and new code to the limits. There are several components to testing the newly implemented FGL. First, this code is in kernel space, which means it is unlike most programs in the fact that in order to run a new kernel the computer is required to reboot. Hence there were two kernels tested, a GIANT_LOCKed kernel and an FGL kernel. The program was not responsible for rebooting and selecting the proper kernel; it just figured out whether it was running an FGL or GIANT_LOCKed kernel, to be able to properly label the data. Sysctl (the main subsystem in this project) is often called from user space through `make` and building programs. Considering this fact, the experiment uses two

standard codebases to build: the kernel itself and the set of "world" programs (all the programs available in the base system of the FreeBSD operating system). The maintenance program is run at a time interval by cron (a time-based job scheduler) to ensure multiple tests were not being run at the same time, since this would corrupt the data being collected.

5.3 KLOG Complications

The kernel event logger (KLOG) was also converted to a fine-grained locking scheme; however, there was a misunderstanding of what originally needed locking. Instead of locking the buffer being filled and jettisoned to print the kernel messages, the locking only protects the device node since the logger just represents a character device (driver) that is prompted by specific in-kernel calls. This fundamental misunderstanding rendered KLOG useless with respect to this project (the lock is only acquired on boot and relinquished at shut-down). The kernel logger was useful however in understanding how locking works in the FreeBSD kernel.

5.4 Improper Usage of FGL

An interesting, but incorrect, usage of FGL would be to use the exclusive lock while reading in addition to writing components of the sysctl data structures. This, in fact, reduces efficiency considerably and increases the amount of time a thread is trying to acquire a lock. To be sure this reduced inefficiency, the same tests used to measure performance between FGL and GIANT were used to test this hypothesis. The tests confirmed decreased inefficiency an significantly increased the total_wait time (time waited to acquire a lock). The reason for this is that using an exclusive lock for reading is effectively the same as using the

GIANT lock; however, since the locking is around the critical sections (instead of just locking sysctl as a whole subsystem) the threads must acquire locks at several different points in the code and thus conflict a lot more. This inhibits concurrency and is counterproductive to the goal of this project.

6 Conclusions and What Really Matters

Once a fine-grained locking scheme was applied to sysctl, the subsystem did produce data that strongly supported the initial hypothesis. There was a half-percent increase in throughput with a maximum time saving of 30 seconds (over a two hour build). The use of FGL helped sysctl do more things in parallel by correctly allowing the program to get the most out of the hardware. The reason that FGL was capable of this is because there is a lot more flexibility when using an FGL scheme over a GIANT_LOCK. Instead of just ensuring mutual exclusion, like the GIANT_LOCK, FGL schemes are capable of controlling more precisely what has access to the data and when. This means the programmer must be aware of what the subsystem is really doing; in the case of sysctl the project improved performance because the FGL scheme was designed around the understanding that sysctl is usually queried for reading instead of writing. As a result, the simple shared exclusive lock was capable of reducing the amount of total wait time to zero even when the amount of queries to sysctl was unrealistically large.

With this power comes great responsibility; the programmer is now responsible for ensuring correctness rather than resting assured that the GIANT_LOCK can prevent race conditions correctly however slow it may be. If more subsystems are converted to FGL schemes there is a potentially large increase in throughput. This is a problem often seen in computer science: increased per-

formance comes with increased complexity. The GIANT_LOCK also ensures against deadlocks for which the programmer also becomes responsible for when using FGL (a **large** part of correctness). With more complex subsystems deadlock and correctness concerns are certainly heightened, but it is clear from the results generated from this experiment that there are benefits to FGL schemes. Even though the sysctl subsystem presented in this project only brought a half-percent increase in throughput these small improvements add up and are even more valuable in the systems programming field because often the systems programs are **always** running. These small increases help to reduce the amount of time the user and cpu are stuck waiting and help the kernel do its job (as a facilitator) much better.

References

[1] E. W. Dijkstra, Cooperating sequential processes. Technological University, Eindhoven, The Netherlands, September 1965. Reprinted in *Programming Languages*, F. Genuys, Ed., Academic Press, New York, 1968, 43-112.

[2] Edsger W. Dijkstra, The structure of the "THE"-multiprogramming system, *Communications of the ACM*, v.11 n.5, p.341-346, May 1968

[3] *The Origin of Concurrent Programming (From Semaphores to Remote Procedure Calls)*, Edited by Per Brinch Hansen, Springer-Verlag New York, INC, 2002.

[4] FreeBSD Manual Pages, Kernel Interface Section 9, `http://www.freebsd.org/cgi/man.cgi?query=locking&apropos=0&sektion=9&manpath=FreeBSD+9-current&format=html`

[5] FreeBSD Manual Pages, Kernel Interface Section 9,, `http://www.freebsd.org/cgi/man.cgi?query=LOCK_PROFILING&apropos=0&sektion=9&manpath=FreeBSD+9-current&format=html`

[6] FreeBSD Manual Pages, Kernel Interface Section 9, `http://www.freebsd.org/cgi/man.cgi?query=sx&apropos=0&sektion=9&manpath=FreeBSD+9-current&format=html`

[7] FreeBSD Manual Pages, Kernel Interface Section 9, `http://www.freebsd.org/cgi/man.cgi?query=mutex&apropos=0&sektion=9&manpath=FreeBSD+9-current&format=html`

[8] FreeBSD Manual Pages, Kernel Interface Section 9,
http://www.freebsd.org/cgi/man.cgi?query=
rwlock&sektion=9&apropos=0&manpath=FreeBSD+9-current

[9] FreeBSD Manual Pages, Kernel Interfaces Section 9,
http://www.freebsd.org/cgi/man.cgi?query=
sema&sektion=9&apropos=0&manpath=FreeBSD+9-current

[10] FreeBSD Manual Pages, Kernel Interfaces Section 9,
http://www.freebsd.org/cgi/man.cgi?query=
condvar&sektion=9&apropos=0&manpath=FreeBSD+9-current