The locking infrastructure in the FreeBSD kernel

Attilio Rao
attilio@FreeBSD.org
The FreeBSD project

Abstract

The increased parallelism introduced by vendors in their architectures for the last years, in order to satisfy the growing request by the user of CPU power, has prodded substantial differences in the software itself. In this paper the support for the multithreading infrastructure that the FreeBSD kernel has adopted, gradually, in the years will be discussed from the developers perspective in order to show strong and weak points of the approach. Along with a technical discussion about the current support, some ideas about how to improve it will be described together with detailed outlines about how to deal with locking problems.

1. Introduction

During the FreeBSD 5.0 development timeframe several efforts have been performed in order to move the kernel from a single-threaded workflow approach to a multithreaded one. As long as this major task (commonly named as **The SMPng project**) has brought a lot of performance-wise benefits it also required to touch heavilly a lot of important core subsystems within the kernel (hacking the scheduler, adding new threads containers, implementing synchronization primitives, moving the consumers to successfully use those, etc.) and offered several challenges for developers to deal with. In order to fully understand them, would be useful to look at how new concepts have been developed, how they influenced the kernel subsystem and what kind of problematics they imposed. A little, technical overview of the situation pre-SMPng seems important in this direction.

Already in the 4-RELEASE serie the kernel was supporting multithreading and preemption as full developed concept, but this scheme was imposing some problems. In particular, if a running thread was getting preempted by another one and then switched back to activity it could have found corrupted or missing resources (structures, lists, etc.). In order to avoid corruptions in a similar context the system had some restrictions like avoiding preemption for threads running in kernel space and disabling interrupts as lock for longer regions of code. Both these protections, however, were totally unuseful in the case of multiple CPUs, where anyone was running its own thread. In such scheme, infact, multiple threads can still access to resources at the same time influencing (and corrupting) the shared objects. In order to solve such problem, only one thread per-time was allowed to run in the kernel as if it was an uniprocessor machine. This had the big unadvantage to loose all the parallelism desired in the kernel space making the presence of multiple CPU, effectively, useless. In this optic the project SMPng was created.

1.1. The SMPng effort

The SMPng project had the aim to make a fully multithreading, preemptive and non-reentrant kernel. It was released for the first time with the FreeBSD-5.0 RELEASE in the 2003 in order to keep up until the FreeBSD-7.0 RELEASE in the 2007.

The initial effort was to offer a serie of suitable locking primitives that could be used in order to lock subsystem and then to switch the **BKL** (Big Kernel Lock), which was preventing more threads from accessing at the same time in the kernel, into a blocking mutex called **Giant**. A new kernel memory allocator was implemented and Giant was removed from the scheduler activities and replaced by a more suitable spin mutex called **sched_lock**. As last step the new concept of ithreads has been implemented.

Through ithreads, any drivers (or interested parties) can install an interrupt handler and assign its own context when running. This was offering the possibility to use sleepable locks to all the interrupt handlers, along with the possibility to perform a lot of interesting operations.

With time, more kernel subsystem was brought out the influence of the Giant lock and their own locking was assigned gradually to them (this included the network parts, the VFS, the VM, etc.).

1.2. The current situation: post-SMPng

Nowadays, all the relevant subsystem of the kernel are nomore under the influence of the Giant lock. The sched_lock is no more present (it has been decomposed in a serie of spinlocks called containers lock) and the ithread mechanism has been completed in order to support a wider and more peforming mechanism called interrupt filtering.

The SMPng project can be considered closed with the RELEASE 7.0 which showed substantial improvements to the kernel scalability.

The main concept behind the FreeBSD locking support is that consumers should know it well and use, where more appropriate, the right primitive in order to solve the locking problem as well as possible. This requires both a good grasp on the problems to solve and on the support provided by the kernel.

2. The locking primitives

There are several locking primitives developed during the SMPng timeframe which can be grouped, basically, into: mutexes, r/w locks, rm locks, waitchannels and semaphores.

The mutexes are primitive offering mutual exclusion for the codepath they want to protect. For example, if a thread owns a mutex and another one contests on it, the latter is unable to go on in the codepath until the former one doesn't release it. This ensures atomicity among several operations of different nature (eg. accesses to different members of the same structure).

The r/w locks (or read/write locks), traditionally are primitives offering 2 different level of acquisition: reader mode and writer mode. If a thread owns the lock in read mode other threads willing to acquire it in read mode will be allowed to run the protected codepath while other (willing to acquire it in write mode) won't. Differently, acquiring the lock in write mode let it behave like if it is a mutex in regard of both kinds of contenders. r/w locks are very useful for protecting resources which are accessed often in distinct, mixed ways (reading and writing).

The rm locks (or read mostly locks) are very similar to r/w but they are developed in order to offer a very small overhead for the read acquisition and a thougher one for the writing acquisition. rm locks are very useful for protecting resources accessed very frequently for reading and very few time for modify.

The waitchannels allow a thread to sleep for a certain amount of time when it wakes up after another thread calling or a timeout expiring. Usually, they are synchronized with a controlling condition through an interlock, which is passed to the primitive serving the waitchannel and released just before to sleep (as well as it is going to be reacquired just after the wake up). They are very useful to synchronize accesses to a resource which is busy for an indefinite timeframe.

Counting semaphores are very similar to waitchannels but for them the controlling condition is always the same: at the very beginning a counter is set to a value bigger than 0 and when a thread acquires the semaphore it does decrease the counter. Once the counter reaches 0 the other threads willing to acquire the semaphore will sleep waiting for a token to be freed by past owners. Please not that a semaphore with a start value of 1 is just a mutex. Semaphores are considered a legacy primitives and they are rarely used nowadays.

A general concept is that all the locks in the FreeBSD kernel can support the *lock recursion*. Lock recursion means that an owner, already holding a lock, can acquire it several time without incurring in problems. This feature is, however, dangerous and discouraged as it is often a symptom of a wrong strategy. So, for all the relevant locking primitives, recursion needs to be selectively enabled (manpages referenced in this chapter can help in the specific case).

However, in order to fully understand strong and weak points for any primitive in FreeBSD it is convenient to identify 3 different categories of primitives based on the behaviour of contesting threads: spinning, blocking and sleeping.

2.1. Spinning category

In this category the only one primitive available is the mutex (mutex(9)).

A spin mutex basically supports mutual exclusion for a code region as if a thread owns it and another contests on it, the latter is going to spin until the former one won't release the mutex. In order words, the contesting thread will remain on the runqueue of the scheduler and will yield code execution until the lock got released. This behaviour has some interesting side effects.

First of anything, a context switch cannot be performed for a thread owning a spinlock otherwise some deadlocks can happen. Immagine, infact, the situation where a spinlock owner gets preempted in favor of another thread with higher priority. If the latter thread wants to acquire the contested spinlock it will start spinning, but it will remain on the runqueue so the scheduler will keep choosing it for running causing a deadlock. Then, another possible deadlock situation cames from the **kernel context** used by interrupts. The top-half of interrupts handler, infacts, runs into a borrowed context, ruled by a particular thread. If this thread owns a spinlock and the interrupt top-half wants to acquire it another deadlock can happen. In order to solve both such situations in the FreeBSD kernel spinlocks need to disable interrupts at all when acquired (and to re-enable again when they are released). This makes their implementation a lot heavier in terms of overhead and consequences (increased interrupt latency) so that often using a spinlock is never a good idea.

There are, though, cases where spinlocks are the only one possible choice. This is the case, for example, of the above mentioned top-half of interrupt handlers running in kernel context. As long as the context has been borrowed they cannot context switch in order to avoid deadlocks so they are forced to use spinlocks.

2.2. Blocking category

For this category there are 3 different primitives available: mutexes (mutex(9)), r/w lock (rwlock(9)), and rm lock (rmlock(9)).

A blocking primitive basically allows contesting threads to be saved into a specific cointainer, to perform some managing stuffs, to be removed from the runqueue and to be switched out from the CPU they are running on. This means that a thread which contests on a blocking lock won't be visible to the scheduler again until the reverse operations will happen: removal from the container, performing managing handover, reinsert in the scheduler runqueues, eventual preemption. The container for blocking primitives is called **turnstile**.

One of the most important managing operations a turnstile performs on its threads is called priority propagation. Priority propagation is a mechanism which try to avoid an high priority thread, blocking on a turnstile to be starved by a lower priority one. For example, if the owner of lock L has a priority A and another thread, with the B priority (with B > A), sleeps on L the former thread will inherit priority B in order to increase its chance to run soon and make the latter thread available as soon as possible. Obviously, the turnstile handover will switch back the owner priority to A. Note that if the owner is sleeping on another turnstile, the priority will be propagated to the owner of the latter turnstile. This means that threads and turnstile makes chains where the turnstiles are links and threads have the duplex role to be alternatively owner and contender (where the head has the highest available priority). This makes clear why the turnstile preserve the concept of owner and they are meaningfull without it: the owner is just the head for a turnstile link. In order to support r/w locks, turnstiles are organized in order to deal with several subqueues (2 at the moment) so that just one turnstile can cater a lock fully. Ideally, however, there is no limit to the number of subqueues a turnstile could handle. For readers, however, no priority inheritance is implemented for turnstiles mainly because it is impossible to track efficiently all the possible owners. A possible workaround for that can be to implement the concept of *owner of record* for the turnstile, where just one reader is tracked. However, it is, obviously, just a temporary workaround in the timespan of the contested reading path.

In any case, performing priority inheritance, tracking and finally performing a context switch (for any single contender) is an high overhead operation, in particular considering that often the time passed between the blocking and the subsequent wake up is not very much. Considering that, an optimization has been implemented in blocking primitives called **adaptive spinning**. Adaptive spinning allows contesting thread to spin on the lock if the owner is actually running on another CPU and so there are good chances the lock is going to be released soon. If the lock owner is not running or suddenly got sleeping the normal behaviour for the contesters happens. Please note that the adaptive spinning mechanism is implemented in the primitives themselves and not in the turnstiles. Finally, it worths tell adaptive spinning works for r/w lock held in read mode too but with the limitation of a timeout as long as readers are not tracked.

Blocking primitives are thought to be the first choice in terms of locking strategies for protecting resources in FreeBSD as long as they are equipped with a full set of optimizations in order to cope with various problems and they don't disable interrupts (or even preemption) in any case. It is obvious that as long as blocking primitives perform a context switch as the very last operation, a thread cannot acquire them while already holding a spinlock.

2.3. Sleeping category

For this category there are 5 primitives available: s/x lock (sx(9)), lockmgr(lockmgr(9)), condition variables (condvar(9)), sleeping points (sleep(9)) and semaphores (sema(9)).

A sleepable primitive basically allows contesting threads to be saved into a specific cointainer, to be removed from the runqueue and to be switched out from the CPU they are running on. The catering container is called **sleepqueue** (sleepqueue(9)).

To a first sight the sleepqueue could seem very similar to a turnstile, but there are a bounch of differences. First of anything, sleepqueues were developed in order to implement only waitchannels (condvar and sleeping points) and this reflects its own structure: sleepqueues do not perform any priority propagation and do not track owners at all (as long as waitchannel are supposed to not have any owner). Then, in order to support efficiently waitchannels, sleepqueues offer the possibility to specify a timeout for the sleeping and a new priority for the threads once it does wake up. They also offer the possibility to catch signals delivered to the sleeping threads and to act appropriately once they wake up. Finally, it offers a full set of mechanisms in order to check sanity of interlocks once they are passed to the primitives. It is obvious that sleepqueues and turnstiles have been developed with specific needs of consumers in mind and they are used to cater totally different actions. More specifically, sleepqueues can be seen as *unbound sleepers* (for long sleeps) while the turnstiles can be seen as *bound sleepers* (for short sleeps). In order to enforce correct locking strategies within the kernel an assertive has been made that no blocking lock (and so no spinlock) can be held while sleeping on a sleepqueue. This happens in order to avoid threads to sleep while holding a mutex and invalidate all the prior work done (priority propagation, adaptive spinning, etc.). This rule can impose some interesting problems that will be explored later.

The sleeping category has the problem to suffer of some legacy features born in the pre-SMPng era (but not only). For example, intially the sleep points were the only way to implement waitchannels even if their interface is messy and not very intuitive. Condition variables were introduced just some years later so that shipping away sleeping points would have result into a not sustainable KPI breakage for thirdy part producers. What should happen, in the future, is to bring the only one feature still missing for condition variables (priority raising) and completely drop the sleeping point, trying to deal with this heavy breakage. New code should, in any case, prefers condition variables for handling waitchannels.

The lockmgr, also, is a legacy product which came directly from the 4.x era. It implements a r/w lock but it also provides a very poor interface which is more similar to a sleeping point than a locking primitive. During the very early times of 5.x era is become very popular and widespread (in particular in the VFS and in the buffer cache) for some of its feature (just like the interlock passing, the possibility to drain consumers, the possibility to specify priority and timeouts to the primitives, etc.) but it is actually deprecated in favor of such primitives like sx lock and condvar. sx locks were created in order to cope, initially, with the impossibility to hold a lock while sleeping. Sometimes it is just too difficult (or simply impossible) to deal with all the sleeping points in a codepath so it is much more convenient to use an sx lock for such codepaths. In any case, as a sleeping primitive, their usage is always deprecated in favor of a better locking strategy involving rwlocks or mutexes.

Finally the other legacy primitives are semaphores. While it is correct to implement them through the use of a sleepqueue, they should be dropped in favor of mutexes and waitchannels in any case. In FreeBSD, just some legacy code use them or paradigms where the value of 0 is passed in order to synchronize startup of some piece of code within a subsystem (aio is an example of that).

2.4. Tips

As the chapter shows, the categories help in identifying a simple hierarchy between locks in the FreeBSD kernel. This helps on various aspects like, for example, the fact that LORs (Lock Order Reversals) and subsequent deadlock can only happen between locks of the same category and it does simplify heuristics and diagnostic tools (for example, spin mutexes have, embdedded, a simple logic in order to catch for deadlocked owner). One of the biggest objection which is often made for the locking primitives in the FreeBSD kernel is that they are too much, sometimes complicated and overlapping. If not considering the legacy supports (sleeping points and lockmgr) and if some simple rules can be followed, building a winning locking strategy is not too difficult.

In particular a consumer should:

- Use a blocking primitive (mutex or rwlock) in the common case
- Use a spinlock if and only if a blocking primitive cannot be used or the path to be protected is extremely little
- Try to deal with unbound sleep, when holding a lock, handling races after the drop instead than immediately switch the locking strategy in order to use an sx lock
- Use an sx lock only when it is necessary to maintain resources consistency across several (and frequent) sleeps
- Avoid completely the legacy support (lockmgr, sleeping point and semaphore)
- Avoid in any case lock recursion

3. Atomic operations

Sometimes it is desired the ability to perform, atomically, very simple operations like, for example, an increment or a bit setting. Simple operations, inside the FreeBSD kernel, can be performed by using directly atomic operations. Such operations are guaranted to be atomic even in the presence of interrupts which means that, when necessary, they disable them (note: that's not the case for a lot of new architectures which directly provide the ability to perform atomic instructions within the ISA, however). Atomic operations are used, mainly, as the ground for building more complex locking primitives (mutex, rwlock, etc) but they can be used also directly by the consumers in an effective way, sometimes helping in building a very low-overhead locking infrastructure. The only side effect is that atomic operations are always limited to operate on the architecture bound so that it is impossible to use them in order to work on large resources safely. Example of atomic operations usage can be located, for example, in the locking primitives code (sys/kern/kern_mutex.c) or in the file descriptor handling code (sys/kern/kern_descrip.c) in order to see how they are used directly in order to make safe code without locking.

Please note that there is no guarantee that the atomic instructions are visible across multiple CPUs. More specifically, there is no guarantee that the atomic operation updates the caches and relevant buffers for all the CPUs involved (or invalidates the cache line where the operation took place) in the SMP system.

Treacting the x86 architecture as a starting point, FreeBSD guarantees that at least operations on 32 bits (and smaller) memory operands can be atomic (like 8 and 16 bits). It is not guaranteed, instead, that all the platform supports the 64 bits sized operations. That is because sometimes it is too difficult to make lightweight atomic operations on 64 bits operands for 32-bits architecture (eg. x86) so, in order to avoid too overheaded operations and forcing consumers in developing a wrong semantic the support is simply dropped.

Sometimes, however, ensuring that an operation will be atomic is not enough in order to have a safe logic. One would need to have guarantees also on the ordering of performed operations. That is when memory barriers came in help.

3.1. Memory barriers

The FreeBSD kernel offers the possibility to perform atomic operations in conjuction with memory barrier. This barrier basically decides the ordering of the atomic operation in regard of other memory accesses in terms of timing. For example, if a CPU performs an atomic operation with a **read memory barrier** it means that all the operations subsequent to this one are going to be visible to the current CPU just after the atomic is completed. On the other side, if a CPU performs an atomic operation with a **write memory barrier** it means that all the operations prior to this one are visible to the current CPU before the atomic takes place. Memory barriers have proven to be very useful in locks building (eg. the lock acquisition will have a read memory barrier while the lock releasing will have a read memory barrier) and in situations where timing is important (context switches). Sometimes atomic operations are also guaranted to not be reordered by the architecture but usually that is not an assumption which can be made within the kernel. Memory barriers, also, specify for the compiler to not reorder or optimize such instructions, resulting in a strong link for code execution.

3.2. Refcounts

Refcounts are a simple and effective way to handle on-the-fly structures which can be freed in any moment. Basically, when a structure is created and consumers work on it asynchronously, they can acquire a reference (which means simply bumping an integer value) for such object so that, a subsequent destroy of it will be stopped in case that the counter is not 0. Often, when dealing with a refcount, you will need to also deal with flags and other members so that a mutex is needed in order to protect those. However, there are simple cases where just a simple add / sub / compare cycle is required to handle the refcount. In these cases the refcount primitive can be used (currently, no manpage is provided for it, unluckilly, but the relevant KPI can be located into sys/sys/refcount.h).

The refcount interface is implemented simply with atomic operations and all the operations are inline so that one can get the fastest primitive available for handling a refcount. That makes it the preferred option when dealing with simple refcounts.

4. Miscellaneous

In conjuction with the above mentioned primitives and mechanisms for building successful locking strategies there are other minor mechanisms that worths to know about in order to deal with multithreading in the FreeBSD kernel.

The first concept to be introduced is the **scheduler barrier**. A scheduler barrier is a mechanism which imposes some constraint on the scheduling of involved threads in a way or another. The schedulers offer mainly 2 scheduling barriers: one controlling threads migration and another one controlling threads execution. A thread which runs a codepath into a section closed between the pair of s**ched_pin() / sched_unpin()** can assume that, while that code executes it will not migrate to run on another CPU. This assumption is very important, for example, when a thread is accessing to per-cpu datas and it wants to maintain a consistency of operations. The second barrier, instead, is rappresented by the s**ched_bind()** function. It does ensure that the provided thread runs only on the specified CPU id. This barrier is particulary useful for kernel threads, created for special purposes, which need to match 1:1 the CPUs present in the system. Sometimes, this mechanism is used, erroneously, in order to force the execution of a thread (created and scheduled very early in the kernel lifespan) on the CPU0 before the other CPUs are started. That is a clear interface abouse which needs to be avoided as much as possible and replaced with effective code. Note that both sched_pin() and sched_bind() are scheduler specific and any new scheduler needs to implement its own in order to maintain the support.

Sometimes, it is desired to disable preemption for some codepaths in order to avoid datas corruptions. In order to do that the pair of function **critical_enter() / critical_exit()** is offered by the kernel. If a codepath is surrounded by such functions it is ensured that the threads preemption will be disabled while it is executed. As long as this feature impacts the whole system (infact, differently by the scheduler barriers, it doesn't just involve the invoking threads, but even the others) it should be used very carefully if not avoided at all. Disabling preemption, by a consumer, can be useful, for example, if the consumer needs to deal with per-cpu datas but it can not tollerate modify to such datas by eventual threads, while performing.

If disabling preemption often is a strong condition, other times threads could have the necessity to disable interrupts at all. The FreeBSD does not support a way to do that in a machine independent way, though an unoffical one can be found. Several places in the kernel are getting used to disable interrupts through the s**pinlock_enter()** interface (and to re-enable them through **spinlock_exit()**). As long as this is an operation which can be potentially very dangerous for performance (if not well used interrupt latency can be increased dramatically) the use of such interfaces is highly discouraged and should be avoided as much as possible.

Sometimes there is the necessity to let run a function (often for managing) on a set of specified CPUs (or all of them). **smp_rendezvous()** and **smp_rendezvous_cpu()** can perform such task by stopping the current activity of the CPU, passing it a setup function, a rendezvous function and a teardown function and waiting for them to be executed. That mechanism is often used in order to perform on-the-fly setups on all the CPU alredy up and running, but it can also be used in order to implement facilities like rmlocks (sys/kern/kern_rmlock.c for a reference).

5. Conclusions

In this paper the full set of synchronization primitives has been presented into an approach which should help the reader to understand what is behind the scene and help him in making the right choice in terms of locking approach choosen. In any case there is still a lot of on-going work for the SMP infrastructure.

For example, legacy locking primitives should be completely removed by all the subsystem and cutted off from the kernel; primitives already existing can be further refined both in terms of performance and usability (just think about back-off algorithms for spinlocks and adaptive spinning parts, new wake up algorithms explored, etc); Giant which still needs to be removed by some subsystems (and ideally to be removed by the whole kernel); locking strategy in the consumers which can be refined and improved.

6. Bibliography

1. Marshall Kirk McKusik and George V. Neville-Neil, **The design and implementation of the FreeBSD operating system**, Addison-Wesley

2. Uresh Vahalia, **Unix internals: the new frontiers**, Prentice-Hall, Inc.

3. Jim Mauro and Richard McDougall, **Solaris internals**, Sun Microsystems Press

4. Curt Schimmel, **Unix Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers**, Addison Wesley

5. The FreeBSD project: http://www.FreeBSD.org