Design Note: Kernel Interrupt and Concurrency Management[†]

Jonathan S. Shapiro, Ph.D.
The EROS Group, LLC

March 18, 2006

Abstract

While Coyotos is an atomic system call design, even the uniprocessor needs to deal with a limited form of concurrency: interrupts. The cleaner, while not concurrent, might also alter the state of objects that have been checked earlier in a given system call path. On SMP machines, there is explicit concurrency to consider as well. This note describes how the kernel implementation deals with these issues.

1 Introduction

There are several sources of concurrency or concurrency-like issues in the Coyotos kernel. On the uniprocessor kernel there are interrupts and cleaning to deal with. On SMP systems, there is true, in-kernel concurrency. This note describes how we plan to deal with each of them.

For SMP readers: note that this document is only intended to cover system designs where there is a single multiprocessor "node." On larger scale (NUMA) machines, more sophisticated techniques are required and it seems likely that a different kernel design is called for.

2 Interrupt Handling

With the exception of the interval timer interrupt and (on SMP systems) the interprocessor interrupt, Coyotos does not handle interrupts directly. Also, the kernel does not support interrupt nesting. Our working assumptions are:

- The longest kernel path is very short (units of microseconds).
- We can therefore defer doing anything about interrupts until we are just about to exit the kernel, when kernel state is not "in flight."
- Since we are going to defer interrupt handling to the scheduler, the kernel does not need to deal with nested interrupts.
- It is preferable, and not much unduly expensive, to prioritize interrupt handlers using the kernel scheduler rather than hardware priorities.
- If it were not for the need to deal with the interval timer interrupt and IPI issues, we could hypothetically run with interrupts entirely disabled in the kernel.

Ignoring the interval timer and the interprocessor interrupt, the primary action taken by the kernel in response to an interrupt is to wake up the relevant driver process by moving it from a per-interrupt stall queue to the ready queue. A preemption occurs only if the awakened process is higher priority than the currently executing process. On hardware platforms where a prompt hardware-level acknowledge is required to avoid interrupt controller lockup, the interrupt handler takes care of this as well.

To interrupt or not to interrupt (the kernel), that is the question. Come to that, it probably *did* sound better in the original Klingon.

2.1 The EROS Design

Historically, EROS took interrupts in supervisor mode because of some legacy hardware issues: certain ISA hardware would get upset if it did not see a prompt interrupt acknowledge cycle from the CPU. There was no need to run the actual handler before exiting the kernel. The issue here was sloppy FPGA designs that would lock up if they didn't get an acknowledge cycle fast enough. I'm not sure if this is still an issue, though it's a sure bet that the week

[†]

THIS SPECIFICATION IS PROVIDED "AS IS" WITHOUT ANY WARRANTIES, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

after I change this behavior somebody will walk up with an embedded card that has this type of bug.

So the EROS strategy was, in effect, to *simulate* a kernel that ran with interrupts disabled. All interrupt handlers proceeded by performing the acknowledge protocol at the interrupt controller level, making a note that the interrupt had arrived, moving processes on the corresponding stall queue to the ready queue, and making a note that a preemption *might* be required because the scheduling environment had changed. Late on the kernel exit path, the kernel would check whether a possible preemption needed to be considered, and would then re-run the scheduler process selection logic in order to decide whether to preempt.

The EROS interrupt handler does not re-enable interrupts prior to return. If it has interrupted the kernel, it simply returns, relying on the kernel to check on the way out. If it has interrupted user mode, it re-runs the scheduler on the way out. The reason it was possible to interrupt the kernel was that we voluntarily enabled interrupts by hand early on the system call entry path. The net effect is that there can be at most one interrupt-related trap frame on the kernel stack at a time. It could be interrupting either user mode or kernel mode, but there was only one.

2.2 Problems with the EROS Design

Problems Conceptually, the EROS design worked pretty well. In hindsight, there were two things wrong with it:

1. **The handler should not touch stall queues.**

   Given that we are going to check for interrupts on the way out anyway, it is possible to update the stall queues there. This as the advantage that stall queue manipulations no longer need to be guarded from interrupt interactions, elminating a possible source of mistakes. In fact, Charlie Landau recently found a bug in the EROS stall queue manipulation code that was caused by exactly this type of interaction.

2. **The handler should perform a non-local return.**

   The EROS handler path explicitly called the rescheduler code if it had interrupted user code. In hindsight this was pointless, because we need to consider a preemption. The smart thing to do in this case would have been to do a non-local return into the re-dispatch logic.

There was a third design issue as well: interrupt disable should not be used for mutual exclusion. This is a common error in uniprocessor kernels. I had always intended to remove it, but never got around to doing so.

2.3 Plan for Coyotos

Plan for Coyotos I'm going to keep the basic EROS design, but in simplified form. On interrupt, the handler will:

- Perform the hardware-level acknowledge cycle, if needed.
- Update a bit corresponding to the interrupt number in a CPU-local `pendingIRQs` bitmask.
- Call `sched_resched()` (a non-local return) if we are returning to user mode, else set the CPU-local `needResched` variable to true.

Note that the *only* points of shared contact between the interrupt handler and the kernel proper are the `pendingIRQs` and `needResched` words. So we have three new requirements on the implementation:

1. It is the responsibility of the kernel to check the `needResched` word late in the path before exiting the kernel and dispatch to `sched_resched()` if it is set.
2. If `needResched` is *not* set at the time of check, interrupts must remain disabled until we are back in user land so that we do not delay an interrupt requiring a reschedule for a full quanta.
3. It is the responsibility of the `sched_resched()` path to wake up drivers as appropriate.

Note that because we invoke `sched_resched()` in response to interrupts, there is no need to do anything special concerning the interval timer.

3 Interactions with the Cleaner

The EROS kernel, and presumably the Coyotos kernel as well, performs demand-driven ageing. The cleaning logic is invoked in response to not having the resource you need. It runs around looking for things that it can free up and either returns to the caller or performs a non-local return. Note that all calls to the cleaner occur in the prepare phase, before the commit point on the path.

One thing that we need to avoid is the situation where the cleaner removes something that has previously been checked by the current kernel path. To prevent this, the EROS kernel used something called a **transient pin**. A transient pin pins a resource in memory only for the duration of the current system call. That is, it renders the resource temporarily exempt from ageing.

For uniprocessors, the mechanism is simple: on every entry to the kernel, a global variable `CurTransaction` is be incremented.[1] Every object has a field `lastTransAction`. To transient pin the object, the code simply writes the value of `CurTransaction` to the object's `lastTransaction` field. If these match, the object is exempt from cleaning. A particular beauty of this design is that nothing ever needs to be undone. When the kernel is next entered, the `CurTransaction` field will be incremented, which releases all transient pins from the previous entry.

This design does not work well for multiprocessors. I have been unable to come up with a comparable design that avoids the need to undo the transient pins, and the pins have to be done on a per-CPU basis (or possibly with a counting semaphore) or races may occur.

One possibility would be to make ageing occur in batches, and consider it a reason to restart the current system call. The ageing code is then exactly as concurrent as any other path, and can be build on whatever locking discipline we require. If the paths are short enough, this might turn out to be the right thing to do, because it is a simpler solution for multiprocessor systems.

I am undecided on all this until I have a better handle on the multiprocessor locking problem.

4 Issues for Concurrency

Implementing concurrency in a microkernel is surprisingly hard. Well, perhaps only if you want to do it well. The "grand kernel lock" approach is probably find for two CPUs.

It is tempting to think that since a microkernel is small and paths are short, the grand kernel lock approach might work better than it did in, say, the Linux kernel. That might be right, but I rather suspect not. We don't have any good data from EROS for how much time is going to be spent in kernel mode on a multiprocessor, but for KeyKOS on a *uniprocessor* the number was somewhere close to 50%. And if that's right, then a grand kernel lock just isn't going to fly beyond two CPUs, and it's not a great solution even for just two CPUs.

It's also worth noting that there is going to be a lot of read sharing: one real benefit of multiprocessing is being able to simultaneously run multiple processes that share an address space on distinct CPUs.

A comprehensive design needs to deal with two classes of issue:

- Managing concurrency on the slow path through appropriate locking and mutual exclusion disciplines.
- Preserving safety on the fast path, which (on its face) doesn't have access to a lot of the key data structures that might need to be locked.

4.1 Things We Do *Not* Need to Guard

It is noteworthy that there are things we do *not* need to protect from cross-CPU interactions. The most obvious one is CPU-local data. There isn't a whole lot of that:

- The `current` pointer is CPU-local.
- The `needResched` and `pendingIRQs` fields are CPU-local.
- The transient mapping space is per-CPU and CPU-local.
- Page directories are per-CPU, but *not* CPU-local.

There are also some things that we don't need go guard because their coherency isn't the kernel's responsibility. In particular, if a user manages to set up two receive data areas that overlap, it is perfectly okay for them to get undefined results. The issues that we are concerned with have to do with management of kernel objects (and their caches) only.

Page Directories A digression about page directories and the transient mapping region is in order.

In order to simplify CPU-local mappings, page directories in Coyotos are private to each CPU. There isn't going to be any TLB sharing across different CPUs anyway, and there are advantages in the kernel if we can do per-CPU mappings for certain kernel data structures.[2] In order to support CPU-private mappings (notably the transient mappings), we decided to go ahead and make address spaces per-CPU. This applies solely to the top level of the translation data structure hierarchy. On IA-32/PSE, for example, the page directory is CPU-local, but page tables are not.

However, an action on one CPU may trigger dependency management logic, and that in turn *can* invalidate an entry in a page directory that is local to a second CPU. The page directories are "local" in the sense that the mappings they

[1] Small detail: to simplify initialization, all transaction values are odd and the increment is performed by two.

[2] Or at least, this was the conclusion that Eric Northup and I reached when we discussed this, and it seems to work reasonably well for xBSD, but as I write this note I am wondering if this remains true on hyperthreaded machines. How do their TLBs do when the master address space pointer is switched? Can we reasonably model them for scheduling purposes as another CPU?

Sigh. Probably not, but this is a bridge we need to burn when we get there. It doesn't look to change the locking requirements, at least.

describe are per-CPU. They are *not* local for purposes of concurrency management.

Transient Mappings The transient mapping region is intended to allow a CPU to temporarily map user pages into the kernel virtual address space. It is the responsibility of the CPU constructing such a mapping to acquire the appropriate lock(s) beforehand, and to ensure that they are not referenced after unlocking.[3] In consequence, the transient mapping region is treated as a CPU-local data structure.

4.2 Reader/Writer Locks and "Upgrades"

Many kernels benefit from locking designs that support multiple simultaneous readers. In the database world, these are sometimes called MROW (multiple reader, one writer) locks. In the linux kernel, this is the `rwlock_t`. There are many places where Coyotos can usefully exploit this type of lock, but there is a complication: preparing capabilities. The act of preparing a capability is not a *logical* mutation of the containing data structure. Unfortunately, it *is* a *physical* mutation, and we therefore need to consider concurrency issues a bit carefully. What we are looking for is a rationale (or at least, a rationalization) for preparing capabilities within a structure that is only "grabbed" for reading. If this is important at all, it will be most important in the PATT structures because of address space sharing.

What we need here is to relax the definition of a "read" lock very carefully, and to justify this we need to step back and look at what a read lock is actually trying to guarantee. A read lock does not mean that we care if writes occur to the object. The actual requirements we care about are the following:

1. If a CPU holds a read lock on an object, no "check" performed on data in that object will be invalidated while the read lock remains held.
2. If a CPU holds a read lock on an object, no data that is *read* from that object will be mutated while the read lock remains held.

Hazard Bits We can (and must) relax the second rule on a special case basis for the capability hazard bit. We want to relax it in this case because (a) we need to be able to build mappings off of a read-only PATT, (b) we need to set the hazard bit in that case anyway, and (c) the hazard bit is an attribute of the capability *slot*, not the capability *value*. The last point is critical: whenever we copy a capability, we zero the hazard bit anyway, which means that "upgrading" the hazard bit value under a read lock is okay.

Note, however, that this relaxation is *only* safe on systems that guarantee data cache coherency. If we don't get that guarantee, we need to manage the issue in software!

Prepare Coherency The rule in the Coyotos kernel is: "always prepare the capability before you do anything with it." Once prepared, a capability can only become deprepared as a consequence of object destruction or pageout. This fact can be exploited, because it leads to the following pair of observations:

1. Because of the "prepare before use" rule, the fact that a capability is *not* prepared is a sufficient proof that no currently active path depends on it.
2. A prepared capability cannot be further "upgraded" once it is prepared.

Taken together, these satisfy the "no mutate after read" restriction; we simply need to wrap a temporary mutual exclusion around the prepare operation itself. I am currently thinking that we *may* be able to do this under a spin lock if the relevant code is written carefully enough.

Destroy/Pageout Regrettably, the capability prepare story only hangs together if we can guarantee that the *target* object will not be destroyed or paged out while a prepared capability remains live in an active kernel path. This means that the act of preparing an object capability requires taking a (potentially upgradeable) *read* lock on the target object. For the majority of objects this is no real difficulty, but taking locks during process operations is moderately tricky.

Summary My sense is that we want to use MROW locks heavily in the Coyotos kernel, primarily driven by the capability prepare logic. The main irritation here is that we need to *undo* all of these locks when performing a nonlocal return. At the moment, I see no good way to avoid keeping a list of taken locks to support this.

One good thing about lock release is that it is all or nothing: either we are abandoning a path and releasing all locks, or we are finishing a path and releasing all locks. This largely eliminates concerns about lock release ordering for the reader/writer locks. As long as the reader/writer locks are re-entrant, we can also stop worrying about lock grab order for *read* locks. We still want to avoid taking multiple write locks on the same object.

4.3 Process Locking

As we will see below, the general rules in the Coyotos kernel are (a) only one write lock may be held on a given

[3] The careful wording here reflects the fact that the transient mapping region is batch-unmapped, so transient mappings may be retained for performance reasons after the mapping itself is logically released.

path, and (b) no locks are held past kernel exit. Processes need to violate both rules, because we cannot simply reach over and modify the kernel state of a process that is presently executing. However, we also need to be able to promptly obtain a lock on a process that is currently running on some other processor. This requires some special handling.

4.4 Stubborn R/W Locks

In conventional kernels, the consequence of requesting an R/W lock and not getting it is that you "sleep" — the requesting kernel thread of control is paused, retaining any locks that it currently holds, and resumes when the desired lock is released.

In an atomic kernel, this approach doesn't work, because it will lead to cascaded blocking. We cannot permit a kernel path to sleep within the kernel, so we must release all of the locks that are currently held. Later, when the lock we wanted becomes available, we will need to re-acquire those locks. Unfortunately, the attempts to re-acquire may fail, and this can lead to cascading rollback.

Ignoring one corner case involving one process manipulating another, the Coyotos implementation does *not* abandon an existing kernel path when it is unable to obtain a lock. Instead, the requesting CPU spins until the holding CPU releases the lock.

5 A Broken Locking Discipline

The big goal in Coyotos is to avoid any situation where we may get into a cyclic dependency. I initially thought that this could be done. It can't, but the idea is seductive enough to be worth documenting. Some ideas are bad enough to justify putting stakes through their hearts. Here was the idea:

Coyotos operations are currently divided into two phases: the prepare phase and the operation phase. These are divided by the commit point. What we do is further subdivide the prepare phase into a marshalling phase and an operation prepare phase. The operation prepare phase is the last phase before stepping across the commit point line. During the marshalling phase, only read locks can be taken, but they may be taken in any order that is convenient. At the transition into the operation prepare phase, all locks taken during the marshalling phase must be dropped, and any subsequent locks must be taken in order of ascending address. Ignoring the process self-lock, at most one write lock may be taken, and at most two locks in total may be taken.

5.1 Single Object Operations

Many Coyotos operations involve only a single capability. This object is ultimately named by a single capability. The kernel invocation as a whole can be divided into two phases:

1. Looking up the capability to the object that we are going to manipulate (marshalling).
2. Invoking that target capability and doing the actual operation.

This class conceptually includes most process operations. While a process has multiple constituents, these constituents are managed for locking purposes as a single unit.

For single-object operations, the marshalling phase is used to perform the capability address space traversal to obtain the capability to the target objects. A local copy of this target capability is made and the marshalling phase locks are now dropped. Next, the per-capability operation handler is entered and the appropriate lock on the target object is acquired.

5.2 Capability Arguments

A larger list of operations involve multiple capabilities, but only one object. For example, storing a capability into a PATT slot requires that we fetch both the PATT capability and the capability to be stored, but it does not require that we *dereference* the capability being stored.

These operations can be treated exactly like single capability operations. The purpose of the marshalling phase is to gather all of the argument capabilities (including the one being invoked). The operation phase then grabs the necessary lock on the target object (in this case, the PATT).

5.3 Multi-Object Operations

A small number of Coyotos operations involve two objects (never more). An obvious example is the "copy capability page" operation, which overwrites one capability page with a copy of the contents of a second. These cases require that we hold more than one lock during the operation phase, and they invariably involve mutating one object. This is the case that motivates dropping the marshalling phase locks just before entering the operation phase.

To avoid deadlock, the operation phase locks must be grabbed in a canonical order. Because acquisition during the marshalling phase is dictated by PATT traversal order, there is no convenient way to order the lock acquisition during the marshalling phase, so if we don't do something unusual we can get into deadlock conditions by holding these locks. Coyotos resolves this by combining three rules:

1. Marshalling phase locks must be read-only.
2. All marshalling phase locks must be dropped before the operation phase.
3. The operation phase may grab a maximum of two locks, only one of which may be a write lock. It must must grab them in address order.

Let's go through the consequences of these rules:

- Locks grabbed by two different CPUs during their respective operation phases cannot involve cycles, because they are acquired in order.
- No attempt during the operation phase to grab a read lock can interfere with any marshalling phase locking, because multiple readers are permitted.
- Once an operation phase has obtained its write lock, it is guaranteed to be able to proceed eventually. If the write lock is the second lock obtained (of two), then the progress is immediate. If the write lock is the first lock obtained, then the only remaining possibility of interference is when the second lock (which must be a read lock) is already locked for writing by some other CPU. But in that case contention will be resolved will be guaranteed by the address order constraint. There may be a chain of CPUs involved, but there will be *some* CPU grabbing a lock at the maximal address (w.r.t. other members of the chain) and this CPU is guaranteed to make progress.

So if we can deal with the special handling required for processes, this *should* work.

Unfortunately, this trick will not work if more than one write lock is required, or if more than two locks total are required, because it can get into iterative conflicts with marshalling phases that are executing on other processors.

5.4 Operations on Processes

The Coyotos kernel API ensures that no operation can involve more than two processes: the invoker and the process named by the invoked process capability. The target process may be running on a second CPU, and in this case the invoking CPU must arrange to temporarily halt the target process. The solution to this is generally to send an interprocessor interrupt to the CPU of the target process in order to get the target process temporarily descheduled.

Unfortunately, this can lead to cyclic interactions. Consider the case where two processes A and B are executing simultaneously on different CPUs and each is attempting to halt the other. Each holds a lock on itself, and each must acquire a lock on the other. Some serial ordering on these operations is required, and one process is going to have to back up in order for either to make progress. Worse, this could all occur over an arbitrary sized ring of processes.

My sense is that these cases are very rare. They occur primarily in debugging situations. Given this, the solution I am inclined to try is the following:

1. Introduce a new operation, `try_write_lock`, that *attempts* to lock the target for writing and returns success or failure. This operation does *not* block if the attempt fails. If this operation succeeds, proceed forward.

   This approach should actually handle the most common cases, because in most situations of cross-process interaction, the target isn't running.

2. Otherwise, the target process is actually executing. Grab a global lock that exists to support this operation. Check if you yourself are already the target of a requested stop. If so, release the lock and back out. Otherwise, see if the target process is already grabbed. If so, block on a queue associated with the target. If neither impediment holds, mark a field in the target indicating that you have requested it to stop, issue an interprocessor interrupt to the target CPU. Now request a write lock on the target in the normal (i.e. blocking) way.

   Note that the global lock prevents cycles in this operation.

I'm confident that there is something basic here that I am missing, but this feels like it ought to work. One issue that I have not yet thought through is the scenario of multiple potential writers ganging up on the write lock. We need to ensure that none of them starve.

Eric Northup offers the interesting thought that all of this works for the capability registers page as well, because events are self-delivered. In order to read from its own capability registers page, a process must grab a read lock on that page on the way into the kernel.

It seems clear that the process coordination case is going to be the most difficult one, and that we will need to refine this.

5.5 The Catch: Return Values

Unfortunately, there is a catch in all this: return values. Kernel operations return register-based data values and (in some cases) capabilities. Both require that the receiving process be locked for writing. When the receiving process is the invoking process, there is no difficulty, but when the receiving process is a third party we may have a problem. It would be pleasant if we could work this out, but at the moment I do not see how.

First, however, note that no kernel operation returns more than one capability, and no kernel operation returns more than two data values. If we add a capability slot into the FCRB structure, we can get away with only needing to lock the outgoing FCRB. We may be able to somehow reason past the lock ordering constraints in that case, but at the moment I do not see how. Perhaps more to the point, I'm unable to make myself reason through it at the moment, and as Charlie Landau pointed out, this whole thing seems likely to be fragile in the face of naïve maintainers.

6 The Coyotos Locking Discipline

On to a mechanism that will actually work. This is the one that (for the present) I plan to actually use.

There are several good points about the mechanism described in Section 5:

- Releasing locks after the marshalling phase significantly reduces the total number of locks involved in any possible contention. This tends to reduce lock contention overall, and it doesn't have any marginal cost worth mentioning, so it seems worth doing.
- At the point where the design grabs operational locks, we have complete knowledge of all remaining locks that will be needed for the rest of the operation. This is also good.
- In between these two phases might be a reasonable place to "pause" when there is contention, provided we can guarantee residency of the target objects or otherwise do something sensible when that residency fails.
- Collision on operation-phase locks should be extremely rare.

6.1 A Simple Solution

Given the points above, my initial thought was to keep the marshalling vs. operation phase distinction, but to view it as an *optimistic* locking strategy. When this strategy fails, more drastic action is required. The more drastic action is to use a global kernel lock.

The global kernel lock effectively controls the overall locking strategy. On every entry into the kernel, this lock is initially acquired in shared mode. A shared acquire indicates that the current kernel thread of control is engaged in optimistic locking, and will restart the system call if any attempt to lock fails.

Execution now proceeds as described in Section 5, with the modification that the operation phase can grab an arbitrary number of locks for either read or write.

If any lock attempt fails, it necessarily fails before the commit point. In this case, the process makes a note that it is failing to acquire and re-starts the system call from scratch. This time, it will acquire the global kernel lock exclusively, and can proceed as if it were running on a uniprocessor.

Scalability This approach, bluntly, sucks. Even if the global lock is never acquired exclusively, it must be touched by every path through the kernel. This means that it has high concurrency even if it has low contention, and will eventually become a scalability limitation. There are some things that we can do about this:

- Break the global lock into per-cluster locks (1 lock per *k* CPUs), and require the contentious case to grab all of these in address order. This reduces normal-case contention.
- Introduce an intermediate strategy wherein the process that failed to lock tries to acquire progressively greater lockout of other CPUs, but not all at once. This might help if we can diagnose the cause of contention at the time it is detected.

Overall, I do not like this solution because all of the tricks I can come up with for making it scale will ultimately fail.

Note that the global lock isn't needed unless we must guarantee individual progress. If statistical guarantees are sufficient, no global lock is needed.

6.2 Rethinking the Operational Locks

It is not clear that we need to restart the system call from scratch if the operational phase locks cannot be acquired. We have, at this point, successfully completed the marshalling unit of operation, and there may not be any reason to give up that work. Suppose we simply spin (perhaps with contention resolution) until we successfully grab all of the operational locks that we require?

It is possible in this case that other operational phases will proceed first, but this is usually okay. The risk is that one of these other operational phases may have the effect of destroying or paging out one of the objects that we are relying on for our own operation.

Note, however, that we can test for this by re-preparing the marshalled capabilities after acquiring these locks. If any of the involved capabilities fails to re-prepare because a page-in is required, we initiate the page-in and restart the operation. All other re-prepare failures must be the result of object destruction. If the current process has been destroyed, we abandon the path. If an argument capability now points to a destroyed object, we proceed as if it had been the Null capability in the first place. If the reply FCRB capability fails to re-prepare, then there is no place to reply to and the reply is dropped, which which is fine. If the capability that we are now operating on becomes Null due to destory, we simply divert to the Null handler (which always works).

My *provisional* belief is that this logic works correctly. It doesn't solve the lock contention problem *per se*, but it at least lets us avoid re-executing the marshalling phase. I'm not sure how expensive the marshalling phase is in practice, but this seems like a good option to document.

6.3 A Refinement

The following conditions hold for the operational lock taking phase:

- Operational locks can be dropped and re-acquired
- The total lock set required for operation completion is known at operation time,
- Operational lock taking does not "hold and wait",

Given these three conditions, it may make sense to avoid the global lock and introduce a serialization lock that is used only for those operations that actually contend. This would not guarantee progress absolutely, but it would ensure progress in practical terms.

Finally, it may be possible to mark the *objects* that are involved in contention such that the processes attempting to lock them "voluntarily" divert to the contented operation path. I need to look into that possibility further.

7 Acknowledgements

Eric Northup consulted on several points in this note.

References

[1] J. S. Shapiro, Eric Northup, M. Scott Doerrie, and Swaroop Sridhar. *Coyotos Microkernel Specification*, 2006, available online at www.coyotos.org.