FSMLABS TECHNICAL REPORT

Against priority inheritance

Victor Yodaiken
Finite State Machine Labs (FSMLabs)

July 9, 2002

1 Inversion and inheritance

There is a mismatch between the properties required from priority driven real-time systems and the property required for mutual exclusion. A priority scheduled real-time system must ensure that the highest priority runnable task can start to run in a bounded time — and the bound needs to be small. A mutual exclusion mechanism must ensure that every task requesting a certain resource wait *as long as it takes* for the task that owns the resource to release it, no matter what the priorities of the tasks. These two constraints can easily conflict causing *priority inversion* — a scheduled task that is waiting for a lower priority task. The classical nightmare case here is when a low priority task owns a resource, a high priority task is blocked waiting for the resource, and intermediate priority tasks keep preempting the low priority task so it cannot make progress towards releasing the resource. Here we have *unbounded priority inversion*. In 1980, Lampson and Redall concisely described the problem with reference to exclusive entry *monitors*.

> *Unless care is taken, the assignment of priorities can be subverted by monitors.[2]*

Lampson and Redall explain that to avoid unbounded inversion, the programmer needs to analyze the program to determine the priority of the highest priority task that locks the resource. The lock operation can then be modified so that any task that holds the lock is temporarily promoted to this priority while it holds the lock. The low priority task is considered to be acting on behalf of the highest priority blocked task and the priority promotion prevents intermediate priority tasks from interfering. This method (now often called *priority ceiling*) works reliably, but has some drawbacks and the analysis can be difficult. *Priority inheritance*[3] promises a solution to unbounded priority inversion without

code analysis. The basic idea of priority inheritance is to provide dynamic calculation of the ceiling priority. When a task blocks on a resource owned by a lower priority task, the lower priority task inherits the priority of the blocking task and continues.

The RTLinux core does not support priority inheritance [1] for a simple reason: priority inheritance is incompatible with reliable real-time system design. Priority inheritance is neither efficient nor reliable. Implementations are either incomplete (and unreliable) or surprisingly complex and intrusive. In fact, the original academic paper presenting priority inheritance [3] specifies (and "proves correct") an inheritance algorithm that is wrong. Worse, the basic intent of the mechanism is to compensate for writing real-time software without taking care of the interaction between priority and mutual exclusion. All too often the result will be incorrect software with errors that are hard to find during test.

Priority inheritance suffers from the drawbacks of all attempts to solve the wrong problem. Synchronization is a price we pay for, sometimes unavoidable, resource conflicts. The best strategy for reducing resource conflict costs is to reduce competition for shared resources. Think of synchronization protocols as analogous to tool check-in/check-out protocols in a repair shop. If the workers are spending too much time arguing over who gets what tool or which workbench, the solution is better scheduling of work, not company wide meetings to resolve each argument. In a companion article[5] I discuss some methods we use to reduce contention in real programs.

2 A quartet of complaints

The purpose for priority inheritance is: *bounding inversion delays without requiring static analysis*.

2.1 Problem: nested critical regions

The first problem is that *nested critical regions protected by priority inheritance locks generate long inversion delays.* **In practice critical regions protected by priority inheritance locks must not contain any inheriting locking operations.**

Suppose T_1 owns mutex m_1 and is waiting for mutex m_2 which is owned by T_2 and so on. If `High` now blocks on m_1 we have to march down the chain promoting each element. If not, `High` would be in danger of unbounded inversion as lower tasks in the chain failed to advance because of intermediate priority tasks. So priority inheritance needs to be a transitive operation. Consider the worst case inversion delay under priority inversion. Ignoring the overhead of the algorithm itself, our task `High` has to wait for every chained critical section to complete! The delay is at least the **sum** of the compute times of the critical

[1] Although it is available on some add-on packages

sections. It's quite possible that the the inversion delays of `High` may be dominated by the critical sections of distantly related tasks. Note that adding task `T` to the system may have dramatic effects on the worst case delays of `High` by connecting `High` to a new chain of linked locks. Suppose, for example, that task `High` can block on a chain of tasks that terminates with a task that blocks on an operating system internal mutex. Now suppose that we add an apparently unrelated task that may acquire this operating system mutex and become the head of its own chain of tasks linked by inheriting mutexes. Suddenly, the worst case inversion delay of `High` is no less than the sum of the critical section compute times of all the tasks in both chains.

Finally consider what happens if a waiting task in an inheritance chain has its timer go off. All tasks down-chain must be disinherited. All of this activity adds to the inversion delay.

2.2 Problem: mixed inheriting and non-inheriting operations

Priority inheritance fails if tasks mix inheriting and non-inheriting blocking operations. **In practice, critical regions protected by priority inheritance locks must not contain any non-inheriting blocking operations either.**

Priority inheritance algorithms assume that each locked resource has a single identifiable owner that can inherit. But many blocking operations do not have single identifiable owners. Consider what happens if a task `T` has a priority inheriting mutex m protecting a critical region that contains a blocking I/O operation on an interprocess communication pseudo-device, such as a pipe or fifo. If task `High` blocks on m and `Low` is waiting for task `VeryLow` to write data into the pipe, then our classic unbounded inversion delay is possible — unless the inheritance algorithm is transitive across different types of blocking operations.

`Low`	lock m	read p			
`High`			lock m		
`VeryLow`				lock $m1$	write p

That is, the inheritance operation must first promote `Low` and then see that `Low` is waiting on the pipe for `VeryLow` and promote `VeryLow`. Even worse, if `VeryLow` is itself waiting on a blocking operation, we have to follow that chain too. However, in general it is impossible to follow the chain because a pipe may not have any owner or it may have many owners and there is often no way to find out. Pipes are simply not suitable for priority inheritance.

Priority inheritance algorithms require that the owner of the lock must release the lock — otherwise the inherited priority is not properly returned. But the standard algorithm for producer consumer involves the producer unlocking and the consumer locking. Suppose that T_1 consumes data passed by T_2 using a semaphore s to synchronize so that T_1 decrements and blocks and T_2 posts. If the T_1 can hold inheriting mutex m when it decrements s, then it is possible

that `High` will block on m and pass its priority to T_1 which is not going to be able to use it. If T_3 preempts T_2, then inversion can arbitrarily delay `High`. Perhaps you think that we can make the inheritance algorithm promote T_2 when it sees that T_1 is blocked on s. But what if there are many producer tasks? What if the producer tasks take turns? Semaphores are not designed for inheritance.

The existence of priority inherit locks makes this error easy to make. There is no good way to statically test that there are no instances of this type of use of locks in the system, and there is no sensible recovery if the error is caught at runtime. Many operating systems run signal handlers in the context of the current task for efficiency. In such a case, a signal handler that releases a lock can cause a fatal error that may not show up on tests.

You might respond that this is hardly fair because priority ceiling has the same trouble. But the static analysis required for priority ceiling is at a level that would uncover our semaphore problem. And who cares about being fair? I care about what happens when the nuclear power plant software needs to shut that valve at once! And I shudder to think of how the programmers designing that plant software may have believed that *they didn't need static analysis because priority inheritance took care of the problem automatically.*

One solution would be to ban inheritance-resistant blocking operations, but POSIX specifies at least the following synchronizing methods that are completely unsuitable for inheritance: semaphores, read/write locks, and blocking I/O. POSIX semaphores are not required to be decremented and incremented in pairs by the same task or even to have an identifiable owner. Read/write locks can have many owners and cannot be efficiently implemented if all owners are identified. And blocking I/O is just hopeless when you think about inheritance.

According to Vahalia [4], the Solaris developers made inheritance "sort-of" work on reader/writer locks. A reader/writer lock allows many readers to hold the lock at the same time but gives exclusive access to a single writer — there are no readers when a writer has the lock. Which reader gets to inherit when a writer blocks? "All of them" is a bad answer because we need to then make acquiring a read lock very expensive (defeating the purpose) and each lock would need auxiliary storage big enough to identify a potentially large number of readers. So the Solaris designers decided to make the first reader the "reader of record" and only promote that one. What happens if the first reader releases the lock before the others? Things don't work, that's what happens: low priority remaining readers still block the writer and they do not inherit. Basically, this trick means that devastating errors will be unlikely to show up in tests.

2.3 Problem: Performance

Priority inheritance worst case performance is worse than the easy alternatives in most cases. **In practice, unless the guarded critical region requires a relatively high compute cost, priority inheritance has guaranteed poor**

performance.

Lampson and Redall's method and all elaborations of that method, including priority inheritance, have a certain level of built in inversion delay while the lower priority task completes the critical region. While you can argue that the lower priority task is doing something on behalf of the higher priority task sharing the resource, the bottom line is that the higher priority task is waiting for the lower priority task. But inheritance increases this built delay.

At its most simple: inheritance costs include:

1. Blocking the higher priority task;
2. Passing the priority down;
3. Restarting the lower priority task;
4. The compute time of the critical section in the lower priority task;
5. Unblocking the higher priority task when the resource is freed;
6. Restarting the higher priority task.

If we allow nesting of priority inherit locks, this can be multiplied by the length of the longest chain.

Now let's consider a simple alternative design: disable all preempts during the critical section. The worst case inversion delay of a task under the "disable preemption" method is the longest compute time of any critical region of any lower priority task in the system. There is no false preempt, no block, no restart, and no transitivity. If the guarded operation is short, such as a link or unlink operation on a queue, there is no doubt that priority inheritance loses. So priority inheritance is only a performance win for task T if the sum of the critical section compute costs plus the overhead of any chain of connected resources is lower than the cost of the most expensive critical resource on some lower priority task that does not belong to any chain including T.

2.4 Problem: Operating System Performance

Inheritance algorithms are complicated and easy to get wrong. **In practice putting priority inheritance into an operating system increases the inversion delays produced by the operating system.**

You could say that many components of operating systems are complicated and easy to get wrong, but it is widely believed that "implementation of the basic priority inheritance protocol is rather straightforward"[3]. In fact, the "basic priority inheritance" algorithm specified by Sha, Rajkumar, and Lehoczky[3] **is** rather straightforward, but it is also incorrect. Basic priority inheritance specifies that when a task releases a lock, it restores its priority to the priority it had before it acquired the lock: inherited priorities are restored using a stack algorithm. Suppose task T locks m_1 and then m_2 and then it inherits a priority on m_1. If T then unlocks m_2 and reverts to the priority it had prior to locking

m_2 it would discard the inherited priority. Instead of using a stack of inherited priorities, a correct inheritance algorithm must keep a list for task of each held lock and for each held lock, there must be a list every inherited priority and when a task releases a lock it must revert to the highest remaining inherited priority. This entire operation must be done atomically to avoid missing inheritances and false inheritances. When a task unlocks an inherit lock it searches through a list of all locks it holds, and for each held lock checks a list of inherited priorities to determine whether what its new priority should be – all atomically. Now consider what happens if we take transitivity into account.

We often want to keep wait-queues in priority order. But priority inheritance can change the priority of threads waiting in queues. As task `High` blocks on l_1 in the example above each task down the queue must inherit and then be moved to the correct spot in its wait queue. Generally we would do this with two queue operations: dequeue and insert where the insert operation on a k length queue can take k steps. In the worst case, as the chain is created task $n-2$ needs to reorder task $n-1$'s wait queue, task $n-3$ needs to reorder both $n-1$ and $n-2$ and, so we may have something like $\sum_{i=2}^{n} i$ queue reorder operations each taking $k+1$ steps. Note that the each descent of a chain and all the queue operations in the descent must be done atomically. Imagine if we protect an operation on a shared queue with a priority inherit mutex and our worst case synchronization cost for protecting the queue operation is an atomic operation consisting of n dequeues and n inserts. Now imagine that task `SuperHigh` becomes runnable at the moment this inherit operation for `High` starts. Even if `SuperHigh` does not share any resources with `High`, it must still wait for this entire atomic operation to complete. So the operating system itself becomes a source of inversion delays for tasks as the operating system atomically carries out the inheritance algorithm on unrelated tasks.

The VxWorks designers originally tried to evade the issue by having a thread retain its highest inherited priority until it released all locks — but this can cause unbounded inversion. Suppose `Low` locks l_1 and then l_2 and inherits from `SuperHigh` on l_2, releases l_2 and continues to use the super-high priority. Reportedly, recent versions of VxWorks have the full algorithm implemented — but see below. The reader should see [4] for a gruesome description of what was needed in Solaris for an implementation that seems to be complete (ignoring the issues raised above on read/write locks and other inheritance unfriendly blocking operations).

Just as chip designers have a certain “transistor budget” that cannot be exceeded without making a chip too expensive to produce, design, and operate, operating system designers have an ”algorithmic complexity” budget. Just as deciding to add decode of complex instructions may mean that the chip has a smaller cache, deciding to add priority inheritance may mean that the operating system can’t use faster data structures.

2.5 Summarizing

Let's summarize the notes:

1. Priority inheritance protected critical sections should not contain inheriting blocking operations.
2. Priority inherit protected critical sections should not contain non-inheriting blocking operations.
3. Priority inheritance adds a significant amount of complexity to the operating system.
4. Priority inheritance protected critical sections should be relatively costly in terms of compute time or they perform worse than the simplest alternative.

As a short comment, if we spot a compute expensive critical region that contains no nested blocking operations, our first thought should not be to cheer that we have finally found a good place to use priority inheritance. Instead we should ask why the task needs to do so much computation inside a critical region.

3 Concluding notes

Avoiding unbounded priority inversion is important, but not not mysterious. Reliable solutions are not easy, but despite claims that actually designing and analyzing code is too hard[1], there is no good substitute. Priority inversion is caused by scheduling when a low priority task holds a resource that may be required by a higher priority task. The obvious solutions are to (1) make the operation using this resource atomic and fast (so there is no scheduling), or (2) remove the contention, or (3) priority schedule the operations. Priority inheritance attempts to provide option (3) as a side-effect. The paper [5] will describe better solutions in detail, but let me conclude with quick examples of all three options.

How do we make an operation atomic? In RTLinux programmers can use the `pthread_spin_lock` operation to disable interrupts and, in an SMP system, set a spin lock. For most queue operations, the sequence `pthread_spin_lock; deq; pthread_spin_unlock` has a far better worst case performance than anything more complex and it is easy to analyze and validate. Even better, we can use an existing atomic operation like `read` and `write` on RT-fifos.

How can we remove contention? The ancient trick of a flip-buffer is always useful. Suppose we have a producer and 5 consumers of sensed data. We can easily reserve, say, 10 buffers. The producer can scan the buffer list looking for a free one and simply drop data if there are no free buffers. The consumers can scan through the buffers using `sem_trylock`. When a consumer is done, it can

mark the buffer free. What do we do if no buffers are available? In most cases, consumers can simply fail if there is no data to consume. Or they can use a counting semaphore to block for data. On RTLinux, the counting semaphore is priority sorted so the highest priority waiter will get the first chance at new data. What about the producer? One strategy is to have the producer do a second scan on a failure and try to grab the semaphore. Because we have 10 buffers and only 5 consumers, we can require that each consumer only hold at most one buffer. The producer can then be assured that there are 5 usable buffers holding stale data that can be overwritten.

And how can we explicitly schedule operations? Suppose we have a database of some sort and we have queries that can potentially take a while to process. Define a structure, say

```
struct request {  myrequest_t request_identifier;
                  pthread_t requestor;
                  sem_t *s;
                  myrecord_t *return_data;
                  }
```

The requester fills out a request, writes the request to a RT-fifo, does a semaphore post on the server thread semaphore and does a semaphore lock to wait. A server process wakes up, orders the queue of requests and completes them in priority order. If we set the server priority to be equal to that of the highest possible requester, then no inversion can take place. For long transaction on behalf of low priority threads, the server can break the transaction into components and keep checking for both new requests and calling `sched_yield` to allow high priority users to advance.

4 Thanks

This note has been in rough draft for many years and has benefited from comments from many people who should not be held responsible for my views or errors. Thanks to Mark Brown at IBM, Prof. Ismael Ripoll at the University of Valencia in Spain, Kevin Danqwardt of K. Computing, and Michael Barabanov and Cort Dougan of FSMLabs for the most recent useful comments.

References

[1] Barbie. Recorded message:math is too hard, around 1995.

[2] B. W. Lampson and D. D. Redell. Experience with processes and monitors in mesa. *Communications of the ACM*, 23(2):105–117, feb 1900.

[3] John P. Lehoczky Lui Sha, Ragunathan Rajkumar. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39:1175–1185, 1990.

[4] Uresh Vahalia. *Unix Internals: The new frontiers*. Prentice-Hall, 1996.

[5] Victor Yodaiken. Synchronization strategies in rtlinux. Technical Report 2FSM2002, FSMLabs, 2002.