# The dangers of priority inheritance

©Victor Yodaiken
Department of Computer Science
New Mexico Institute of Technology
`yodaiken@cs.nmt.edu`
DRAFT (corrections and comments appreciated)

The recent use of *priority inheritance* (PI) to repair a software failure on the Mars Lander has focused positive attention on this technique without much reference to its serious, and dangerous, drawbacks. While some limitations of PI have certainly been mentioned in the real-time literature, the method is so fragile, so prone to serious error, and so unnecessary, that it seems important to make its limitations rather more clear. It's also worth noting that a published proof of the correctness of priority inheritance [RS90] is wrong.

## 1 Priority inversion

Certain real-time applications can be implemented by in terms of prioritized concurrent tasks where tasks with tight deadlines are given higher priorities than tasks with slacker deadlines. Real-time scheduling disciplines, such as rate monotonic and earliest deadline first, attempt to ensure that the most urgent tasks have first claim on the processor. All of these real-time priority based scheduling disciplines fail if a low priority task can significantly delay execution of a higher priority task. But, in many applications, there are resources that must be shared among tasks in a way that prevents more than one task from using the resource at any moment. *Priority inversion* is the term used to describe a situation where a task is waiting for a lower priority task to free a shared, exclusive use, resource. Clearly, a priority based real-time system cannot tolerate significant periods of priority inversion.

The most widely known (and taught) technique for protecting shared resources is via semaphore guarded critical regions. Sadly enough, semaphore guarded critical regions introduce exactly the sort of delay that priority scheduling cannot tolerate. The obvious problem situation arises when task must wait for a lower priority task to relinquish a shared semaphore. A secondary problem situation occurs when a low priority task holding a semaphore that is shared with a high priority task is preempted by a third task with intermediate priority. The third task prevents the low priority task from advancing and releasing the semaphore. In this case, the high priority task is effectively blocked by the third task. This is the error that caused program failure on the Mars Lander.

## 2 Priority inheritance

Priority inheritance mechanisms are intended to prevent priority inversion by adjusting the *effective priority* of a task $\mathsf{T}$ whenever a higher priority task is suspended waiting for a semaphore owned by $\mathsf{T}$. That is $\mathsf{T}$ *inherits* priorities from tasks waiting for $\mathsf{T}$ to complete its critical region. Inheritance is simple enough conceptually, but in practice it is rather more complex. Since there are many flavors of PI, I'll begin by making a precise definition of simple priority inheritance (SPI) and then consider enhancements.

**Definition. 1 The simple priority inheritance algorithm.**

*When a task* $\mathsf{T}$ *attempts to acquire a semaphore that is held by a second, lower priority task* $\mathsf{T}'$*, the effective priority of* $\mathsf{T}'$ *is increased to equal the priority of* $\mathsf{T}$. *When a task frees all semaphores it returns to its initial priority.*

This method is essentially the method implemented in VxWorks and other RTOSs and differs significantly from the algorithm that [RS90] calls *basic priority inheritance.* Unfortunately, SPI does not solve priority inversion.

In the following, suppose we have tasks $\mathsf{T}_1, \ldots, \mathsf{T}_n$ where the initial priority of $\mathsf{T}_i$ is $p_i$ and $p > p'$ means that $p$ is a higher priority. Suppose we also have semaphores $A, B, C \ldots$.

1. Direct priority inversion is unaffected: if $\mathsf{T}_j$ acquires semaphore $A$ and $\mathsf{T}_i$ with $p_i > p_j$ becomes active and attempts to acquire $A$, $\mathsf{T}_i$ must still wait until $\mathsf{T}_j$ releases $A$.

2. As with all semaphores, deadlocks are easily constructed if semaphores are acquired in the wrong order. For example:
   $\mathsf{T}_2$ obtains $A$;
   $\mathsf{T}_1$ with higher priority preempts, then obtains $B$ and then blocks on $A$;
   $\mathsf{T}_2$ then blocks on $B$.

3. Because a task retains the highest inherited priority until all semaphores are released, the algorithm can *cause* priority inversion. For example suppose $p_4 < p_3 < p_2 < p_1$:
   $\mathsf{T}_4$ obtains $A$ and then $B$ ;
   $\mathsf{T}_1$ preempts, then blocks on $B$ and passes its priority to $\mathsf{T}_4$;
   $\mathsf{T}_4$ releases $B$ and continues to run at $\mathsf{T}_1$ priority.

4. Chained semaphore usage can still result in priority inversion. For example suppose $p_4 < p_3 < p_2 < p_1$:
   $\mathsf{T}_4$ acquires $A$ ;
   $\mathsf{T}_3$ preempts, acquires $B$, and then blocks on $A$;
   $\mathsf{T}_4$ becomes the active process;
   $\mathsf{T}_1$ preempts, then blocks on $B$ passing its priority to $\mathsf{T}_3$ with no effect ;
   $\mathsf{T}_2$ becomes the active process, blocking progress for $\mathsf{T}_4$ and thus, indirectly, for $\mathsf{T}_1$.

## 3 Basic priority inheritance

Sha *et al*[RS90] define what they call *Basic priority inheritance* (BPI) that differs from SPI in the following.

1. When a task relinquishes a semaphore, its effective priority is restored to the priority it had before acquiring the semaphore.

2. Inheritance is transitive. When a task $\mathsf{T}$ blocks on a semaphore owned by a task $\mathsf{T}'$ any task that is blocking $\mathsf{T}'$ must also inherit the priority of $\mathsf{T}$ and so on. The transitivity prevents chained priority inversion.

Sha *et al* (and also [RSLR95]) note that BPI is still subject to deadlocks and that it is possible for a high priority task to have to wait for multiple chained critical areas to complete (the transitivity causes chained delays). Chained delays are exceptionally difficult to calculate and can be very significant, but these are not the only problems with BPI. In fact, Sha *et al*'s proof that BPI has bounded delays is incorrect.

Sha *et al* specify that "when [task] J exits a critical section, it resumes the priority it had at the point of entry into the critical section" (p 1177). Suppose a task with priority $p$ acquires $A$, acquires $B$, and then frees $A$. Suppose that the task inherits priority $p_A$ after acquiring $A$ and then inherits a higher priority $p_B$ after acquiring $B$. When the task frees $A$, what should be its effective priority? BPI restores the priority to $p$. But this defeats priority inheritance as after releasing $A$ the task is no longer running at the inherited priority $p_B$ even though it holds $B$. Sha *et al* avoid this problem by assuming that all critical regions are nested with no overlap. That is, they require that critical regions follow a strict stack discipline. For the moment, let's put aside the question of whether this is a reasonable assumption and consider what happens if the assumption is valid. It turns out that if the preempting tasks preempt and pass inheritance in the wrong order, we can get the same effect in any case. Recall that if $i > j$, then the priority of $T_i$ is lower than the priority of $T_j$. Suppose that $T_4$ with priority $p_4$ acquires $A$ and then $B$. Suppose that $T_2$ preempts and blocks on $B$, passing its priority $p_2$ to $T_4$. Now suppose $T_1$ preempts and blocks on $A$ passing its priority $p_1$ to $T_4$. When $T_4$ now releases $B$ and "resumes the priority it had at the point of entry into the critical section", its priority becomes $p_4$. Thus, $T_1$ is now suspended on the wait queue for $A$, waiting for task $T_4$ with priority $p_4 < p_1$ to complete. If $T_2$ now becomes runnable again, $T_2$ will preempt $T_4$ and $T_1$ will be waiting on a lower priority task that does not necessarily need to acquire any shared semaphores. Thus, Lemma 1 in [RS90](p 1177) is not correct and similarly for all the results that follow from Lemma 1 in particular the claim that there is a bound on the waiting time for a task under BPI.

We can try to repair BPI by requiring that when a task frees a semaphore, its priority is set to the highest inherited priority for any remaining held semaphore. To implement such a policy, each task would need to maintain a set of pairs *(semaphore, inherited—priority)*. When a task passes its priority to the task blocking it on semaphore $A$, a pair $(A, p_A)$ would be added to the set of the owner task – or the current pair $(A, p)$ would be updated with the new priority. When a task released a semaphore it would have to remove the pair with that semaphore as a member and set a new priority to be the highest remaining priority in the set. The operation of searching and updating the set and resetting the new priority seems as if it must be done atomically.

```
UpdateInherit(task T, semaphore A, priority p){
     If( InheritedPriority(A,T) == null)
         InsertInheritedPriority(A,p,T);
     If( InheritedPriority(A,T) < p )
         ReplaceInheritedPriority(A,T,p);
     If( p > EffectivePriority(T))
         SetPriority(T,p);
     }
```

```
UpdateRelease(semaphore A, task T){
      RemoveInheritedPriority(A,T);
      SetPriority(T, MaxInheritedPriority(T));
      }
```

Now consider how transitivity must be implemented. A blocking task needs to recursively descend the chain of semaphore owners — possibly traversing the entire set of tasks in the system. The traversal might work like this:

```
Inherit(semaphore A,priority p){
      UpdateInherit(s->owner,A,p)
      If WaitingSemaphore(s->owner) != null)
         Inherit(WaitingSemaphore(s->Owner),A,p)
      return;
        }
```

The entire traversal must be done atomically to prevent changes in the linked list of waiting tasks and semaphores.

I hope that this fixes the problem with BPI and rescues Lemma 1 of Sha *et al*, but correctness is, clearly, an elusive property in this area. In any event, the solution is not very satisfactory for a couple of reasons:

1. The problem of critical regions that are not properly nested is still being solved by pretending it does not exist. Validating this property in complex systems is a daunting prospect and there is no obvious run-time solution. What can one do when an error is detected?

2. When we calculate the possible delays of a task T with priority p, we need to sum all the delays of every task with priority greater than p, plus every critical area of every task that, directly or indirectly, shares semaphores with any of those tasks, plus the longest chain of critical areas of tasks that share, directly or indirectly, any semaphores with T.

3. Transitivity needs atomic operations on at least one linked list. My experience is that most critical regions involve operations on linked lists. Thus, we invoke an operation that does an atomic operation on a linked list L in order to safeguard an operation on a linked list $L'$. This should make one ask whether it would not be better just to do the operation on $L'$ atomically.

4. The entire system depends on an assumption that semaphores are always released by the same task that acquires the semaphore. In RT code, however, it is very common to see semaphores released by interrupt handlers or by producers of data as a way of controlling data flow. For example, suppose we have a device D that is guarded by a semaphore A so that a task wanting to use D must acquire A. If D can generate a completion interrupt, the standard solution would be to make a requesting task acquire A and then have a interrupt handler release A to the next task. Or consider

this code.
A and B are initially down with $T_x$ owning A and $T_y$ owning B.
$T_x$: `while(1){put data in buffer;up(A); compute; down(B);};`
$T_y$ `while(1){down(A); consume data in buffer; up(B); compute;};`

## 4 Improvements

As Sha *et al* note, Lampson and Redell[LR00] considered the problem of priority inversion for monitors and suggested that a task entering a monitor promote its priority to the highest possible priority of any task that might use the monitor. The same proposal can be applied to semaphores with an interesting result: either a task that holds a shared semaphore is not permitted to become unrunnable (*e.g.* by suspending itself on another shared semaphore) and no task will ever be queued waiting for a shared semaphore, or tasks holding semaphores can become unrunnable and the method fails to prevent priority inversion.

- **Case 1**. Suppose that T acquires semaphore A and while T holds A, it has a priority greater than any other task that shares A and so no other task that would attempt to acquire A can run. It follows that the wait queue for A is always empty. In other words, we have converted semaphores into monitors.
- **Case 2**. If T can suspend, then nothing is gained by inheritance. Suppose T suspends on a second semaphore B. At this point, if the owner of B suspends: any A task can run, but none can progress. Again, transitivity is needed in order to avoid some of the problems of chained semaphores.

## 5 Some discussion

A semaphore guarded critical region makes an arbitrary code fragment become a critical timing pathway for every task that shares the semaphores directly or indirectly. Furthermore, nesting of semaphores introduces many opportunities for deadlock and unpredictable timing interactions. We see a steep increase in complexity, and thus possible logical and implementation errors as more and more improvements are added to priority inheritance schemes.

Given its complexity, the appeal of priority inheritance is quite mystifying. If we look at the alternatives, the mystery only deepens. Here are three obvious alternative methods of sharing access to exclusive sue resources.

1. Atomic operations. If the operation on the resource takes little time, we can simply offer atomic operations that cannot be pre-empted. Many operations on complex data structures can also be factored into small "commit" operations that need to be atomic and more complex operations that can be performed in non-shared space. For example, to queue an element in a linked list, a low priority task can allocate a new element, fill in the pointers on this new object and, only then, lock the queue to change the head pointer. For many operations of this type, we can perform optimistic

operations: assuming that there will be no preemption, and then abort if preemption is detected.

```
enq(queue *q, data *d){
      e = allocate_new_element;
      copy_data(e->data,d);
   retry:
      temp = q->change_count;
      search queue for place to put e
      e->next = right_place->next;
      disable_preemption();
      if (q->change_count != temp){ /*optimize common case */
             enable_preemption();
             goto retry;
             }
       right_place->next = e;
       q->change_count = temp+1;
       enable_preemption();
     }
```

2. Use lock free data structures such as those described in [Her91, PMI88] and as used in double buffering or similar shared memory techniques.

3. Servers. If the operation on the resource consumes a significant chunk of time, we can centralize operations in a server task or server co-routine. The server can prioritize and in many cases easily abort low priority requests. We can even use semaphores to enq requests.

```
request:
        Atomic_Enq(serverq,request);
        down(serverwaitq[mytaskid]);
        ....

Serverloop:
       request = BlockDeqHighestPriority(serverq);
       do request;
       up(serverwaitq[request->task->id]);
       goto Serverloop;
```

To make the discussion more concrete, let's consider the Mars Lander problem itself. According Glenn Reeves[Ree98] the problem was buried inside the VxWorks I/O subsystem. A low priority task (the `ASI/MET` task) was connected to the high priority `bc_dist` task via a VxWorks IPC mechanism The `ASI/MET` task made a `select` call that, within VxWorks, invoked a semaphore protecting the file descriptors under the select. One of these file descriptors was for the IPC pipe between `bc_dist` and `ASI/MET`. Before the semaphore could be released, the `ASI/MET` task was preempted by some medium priority tasks. The next invocation of the higher priority `bc_dist` task then stalled when it attempted to send more data on the IPC pipe.

From Reeves summary, it appears as if the ASI/MET task broke the conventions of the system by making use of the pipe mechanism in place of the double-buffered shared memory used for by all other tasks. So the ASI/MET task made use of VxWorks IPC, and VxWorks IPC made use of a semaphore. This, in my humble opinion, amply illustrates the dangers of semaphores, buried in lower level code and producing a critical path as a side effect. Note that double-buffered shared memory, or a lock-free queue, or even a queue with atomic `enq` and `deq` operations would have avoided this error. To "fix" the error, NASA programmers enabled VxWorks priority inheritance for the `select` code. As noted above, such a fix does not necessarily fix anything, and Reeves points out that the highly skilled programmers at NASA, with access to VxWorks source, were not very confident about the effects of this change. My surmise is that the fix worked for two reasons: (1) the system generally avoided semaphores, and (2) the engineers had good luck.

## References


[Her91] M. P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages*, 13:124–149, 1991.

[LR00] B. W. Lampson and D. D. Redell. Experience with processes and monitors in mesa. *Communications of the ACM*, 23(2):105–117, feb 1900.

[PMI88] C. Pu, H. Massalin, and J. Ioannidis. The synthesis kernel. *Computing Systems*, 1(1):11–32, 1988.

[Ree98] Glenn E. Reeves. What really happened on mars. In *RISKS Forum (19.54)*, risks@sri.com, jan 1998.

[RS90] Sha L. Rajkumar R. and S. Sathaye. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39:1175–1185, 1990.

[RSLR95] Ragunthan Rjakumar, Lui Sha, John P. Lehoczky, and Krithi Ramamritham. An optimal priority inheritance policy for synchronization in real-time systems. In Sang H. Song, editor, *Advances in Real-Time Systems*. Prentice-Hall, 1995.