Reducing Priority Inversion in Interprocessor Synchronization on a Fixed-Priority Bus*

Chi-Sing Chen (cschen@cs.fsu.edu)
Pratit Santiprabhob (pratit@cs.fsu.edu)
T.P. Baker (baker@cs.fsu.edu)
Department of Computer Science B-173
The Florida State University
Tallahassee, FL 32306-4019

August 31, 1991

Keywords: Ada, real time, multiprocessor, kernel, mutual exclusion, priority inversion, synchronization.

*This work is supported in part by the Florida High Technology and Industry Council.

Reducing Priority Inversion in Interprocessor Synchronization on a Fixed-Priority Bus

1 Introduction

This paper describes an interprocessor synchronization algorithm in which the priority of the requesting process as the major consideration for acquiring a shared resource. The algorithm was developed to implement an Ada Run-Time Kernel (RTK), on a multi-processor which has fixed processor priorities. The RTK supports a collection of *virtual processors* (VP's) that execute on a collection of processors with some shared memory.

A key feature of the RTK execution model is that all resources are scheduled according to a unified priority scheme, in which VP's, interrupt handlers (IH's), and locks protecting nonpreemptable resources all have associated priorities. The *active priority* of a VP is the maximum of its own priority, the priorities of any locks it is holding, and the priorities of any IH's it may be executing. This priority inheritance strictly limits the duration of priority inversion[4], and permits Rate Monotone schedulability analysis. Integration of VP and IH priority enforcement also solves several well known synchronization problems between scheduled processes and asynchronous interrupt handlers[1, 3, 5]. More detail about the RTK can be found elsewhere [2, 6].

The key issue in implementing the RTK on a multiprocessor system is to provide an interprocessor locking mechanism that respects VP priorities. Such a locking mechanism has been developed, that operates on a system consisting of several Motorola MVME 133a processor boards connected by a VME bus. These processors each have *local* memory. They can access local memory without recourse to the VME bus, and can access the local memory of the other processors remotely, through the VME bus. There is also *shared* memory, which can be accessed by all of the processors over the VME bus.

In the following sections, we describe the difficulties presented by this machine architecture, and an algorithm that works around these difficulties.

2 Difficulties Imposed by the Hardware

2.1 Slow Bus Speed

In order to execute a VME bus cycle (read, write, or interrupt acknowledge), the processor which wants to access data through the bus must obtain the bus mastership. This results in very slow access time, compared to that for local memory. In order to enforce the priority scheme of the RTK, some use of

global data structures is unavoidable, but an efficient locking mechanism must keep accesses to this global data as infrequent as possible.

2.2 Fixed Bus Priority

The VME bus arbitrates between processors according to a fixed priority scheme, based on board position. We will call this fixed bus priority of each processor its *privilege*, to distinguish it from the active priority of the processor, which is that of the VP or IH that happens to be executing. Mastership of the bus will be always granted to the processor with the highest privilege. This raises a serious bus contention problem, as well as potential priority inversion situation, when shared variables are used for inter-processor locking. To solve the bus contention problem, it is necessary that busy-waiting on shared variables be avoided. This means that whenever a processor attempts to gain a lock and fails (using a test-and-set or compare-and-swap instruction), it busy-waits on a local variable, instead of a shared variable, so that processors with lower privilege have a chance to access the global data through the VME bus.

2.3 Priority Inversion

Priority inversion is defined as a condition in which a high priority process is delayed by a process with lower priority. This may occur due to the VME bus's fixed processor privilege scheme. It is possible that a high priority VP, which is executing on a low privilege processor, is blocked from accessing shared data through the bus by another VP with lower priority residing on a higher privilege processor. This hardware-level priority inversion is unavoidable with the VME bus.

To minimize the effects of this priority inversion on VP scheduling, interprocessor lock acquisition is implemented in two stages. The first stage involves acquisition of the *outer lock*, of which there is only one, so the length of time it is held must be kept very short. The second stage involves acquisition of an *inner lock*, of which there may be several. The outer lock protects a processor that is attempting to acquire or release an inner lock, from interference by other processors. This enables the inner lock to be granted according to VP priority, and the outer lock to be passed on to the next processor also according VP priority.

2.4 Race Conditions

Because acquisition of an inner lock is not supported by an atomic action at the hardware level, care must be taken to avoid race conditions, which can result in deadlock. We encountered one such problem in the design of our algorithm. When releasing the outer lock we want to grant the outer lock to the processor that has highest VP priority. This is implemented by scanning a global data structure (`Waiting`) to find the waiting processor, if any, that has the highest VP priority. A dangerous race may occur when processor A records its intention to wait in `Waiting` while processor B is scanning to see whether it should pass on the lock, or just release it. If processor A records its intention to wait after B has scanned past

it, but tests the lock before B releases it, A may wait forever.

To solve this problem, the processor releasing a lock must recheck for waiting processors after releasing the lock. In our algorithm, this is implemented using another global variable (`Scanning`). When the VP is releasing the outer lock, it will set `Scanning` to `true` first and then scan `Waiting` for the processor with the highest VP priority. Then, after the outer lock is released or passed, it will check `Scanning` again. Since every processor has to reset `Scanning` to `false` before it competes for the outer lock, a new arriving processor which wants to grab the outer lock will be detected. In this case, the processor that has just released the outer lock will try to get it back, by competing with other processors in the normal way; if it fails to acquire the lock immediately (i.e. the outer lock has already been seized by another processor), it will either continue into its critical section (if it has the inner lock) or start busy-waiting on a local variable; otherwise, it will re-scan the `Waiting` table and try to catch the late arrival.

2.5 Unbounded Waiting

Making the priority of a VP the major consideration for acquiring a lock may cause unbounded waiting. Since there may always be a VP with high priority requesting the lock, a VP with low priority may never have a chance to get it. Unbounded waiting is traditionally viewed as a problem in concurrent programming, when it happens arbitrarily, but here it is only due to strict enforcement of priorities. This is an unavoidable property of priority scheduling. In a priority-based real-time system, the only way to avoid unbounded waiting due to priority enforcement is to control the load, so that there is adequate time for higher priority processes to complete, allowing the lower priority processes a chance to execute.

3 The Algorithm

In this section, we describe the locking algorithm in detail. There are two routines, `Spinlock` and `Releasespin`, which a VP calls when it wants to seize or release in interprocessor lock.

3.1 Data structures

The following data structures are in shared memory, accessible by all processors.

```
type Wait is
  record
    Want : Boolean:= false;
    Prio : integer:= 0;
    Lock : Lock_ID;
  end record;
GL         : Boolean:= false;
Waiting    : array(Processor_ID) of Wait_type;
Inner_Lock : array(Lock_ID) of Boolean:= (others=> false);
Scanning : Boolean:= false;
```

GL represents the outer lock. If Waiting(P).Want is true, it indicates that processor P wants to acquire GL. Waiting(P).Prio is the priority of the currently executing VP on processor P, if it is waiting for a lock; otherwise it is zero. Waiting(P).Lock indicates which inner lock the processor is waiting for. Inner_lock(L) acts as the inner lock. Scanning indicates whether it will be necessary to re-scan Waiting when the processor holding GL is trying to find the waiting processor that has the highest priority.

The following variables are replicated, one copy in the local memory of each processor:

```
Local_Wait : Boolean:= false;
Self: Processor_ID:= ...ID of this processor...
```

3.2 Spinlock

A simplified rendition of the code for Spinlock is presented below. This code uses the following subprograms and variables.

1. Local_Wait is the variable that the local processor will spin on when it is busy-waiting. Self is the ID of the local processor.
2. Test_and_Set(V) is an atomic function that tries to set its argument variable to true, returning true if it succeeds and false otherwise.
3. Reset_Remote_Wait(P) is an operation that sets the value of the Local_Wait variable on processor P to the value false.
4. Choose_Highest(L) is a function that searches the Waiting table for the highest-priority processor that is waiting for the inner lock L, and returns zero if there is none.
5. Choose_Highes_GL is a function that searches the Waiting table for the highest-priority processor that is waiting for GL, and returns zero if there is none.

```
procedure Spinlock(Caller_Priority: VP_Priority; L: Lock_ID) is
    Choice: Processor_ID;
begin
    Waiting(Self).Want := true;
    Waiting(Self).Prio := Caller_Priority;
    Waiting(Self).Lock := L;
    Scanning           := false;
    Local_Wait         := true;
    if Test_and_Set(GL) then
          while Local_Wait loop null; end loop; -- (1)
    end if;
    -- Self is holding GL and Prio/=0 or holding Inner_Lock(L) and Prio=0
    if Waiting(Self).Prio /= 0 then
          if Inner_Lock(L) then
                Choice:= Choose_Highest(L);
                Waiting(Choice).Prio := 0; Reset_Remote_Wait(Choice);
          end if;
          Waiting(Self).Want := false;
```

```
          loop  Scanning := true;
                Choice:= Choose_Highest_GL;
                if Choice = 0 then GL:= false;
                else Reset_Remote_Wait(Choice);
                end if;
                exit when Scanning or else Test_and_Set(GL);
           end loop;
           while Local_Wait loop null; end loop; -- (2)
    end if;
end Spinlock;
```

When the VP which is currently executing on a processor wants to acquire a lock, it needs to compete for the outer lock with the VPs that are currently executing on the other processors, by applying the atomic instruction `Test_and_Set`. The `Test_and_Set` instruction allows only one VP to grab the outer lock; so the others will then have to busy-wait on the local variable `Local_Wait`. If the VP succeeds in seizing the outer lock, it will check the availability of the inner lock. If the inner lock is free, then all the processors will be scanned and the inner lock will be passed to the VP with the highest priority (by remotely resetting the `local_wait` variable of that VP's processor). Note that if a VP with the highest priority tries to seize the outer lock at this moment, and goes into the busy-wait at (1), this VP will come out of the busy-wait already holding the inner lock that it wanted; it will know it has the inner lock, since its waiting priority will have been reset to zero.

After the inner lock is granted, either to the VP itself or a VP on another processor, the VP will scan the global `Waiting` array again trying to find the VP which is waiting for the outer lock and has the highest priority. If the re-scan indicator `Scanning` indicates that no other VP has tried to seize the outer lock during the scanning operation, the VP will jump out of the loop and either busy-wait wait at (2) or go into its critical section (if it has seized the inner lock and reset its own `Local_Wait` variable).

3.3 Releasespin

```
procedure Releasespin(Priority: VP_Priority; L: Lock_ID) is
     Choice: Processor_ID;
begin
     Waiting(Self).Want := true;
     Waiting(Self).Prio := Caller_Priority;
     Waiting(Self).Lock := L;
     Scanning           := false;
     Local_wait         := true;
     if Test_and_Set(GL) then
           while Local_Wait loop; null; end loop; -- (3)
     end if;
     Waiting(Self).Prio := 0;
     Choice:= Choose_Highest(L);
     if Choice /=0 then
           Waiting(Choice).Prio := 0; Reset_Remote_Wait(Choice);
     else Inner_Lock(L):= false;
     end if;
     Waiting(Self).Want := false;
     loop  Scanning := true; Choice:= Choose_Highest_GL;
           if Choice = 0 then GL:= false;
```

```
            else Reset_Remote_Wait(Choice);
            end if;
            exit when Scanning or else Test_and_Set(GL);
        end loop;
    end Releasespin;
```

A simplification of the code for `Releasespin` is presented above. As in `Spinlock`, the VP which wants to release a lock must compete with the other VPs. The inner lock is granted to the waiting VP with the highest priority, or released. The procedure for releasing the outer lock is the same as in the `Spinlock`, to avoid deadlock due to the race described in Section 2.4.

4 Conclusions

In this paper, we have described an algorithm designed to provide interprocessor synchronization with bounded priority inversion, for a system with fixed processor bus priorities. We have tested this algorithm. We next plan to measure its performance, and see whether we can speed it up by using a faster data structure for enqueuing waiting processors and VP's,

References

[1] Ada 9X Project Office, "Draft Mapping Document", Ada 9X Project Report, Office of the Under Secretary of Defense for Acquisition (February 1991).

[2] T.P. Baker and Offer Pazy, "A Run-Time Kernel Proposal", ACM Ada Run-Time Environment Working Group, to appear.

[3] T.P. Baker, "Comment on: 'Signaling from within Interrupt Handlers"", *Ada Letters XI,1* (January/February 1991) 17-18.

[4] J.B. Goodenough and L. Sha, "The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High Priority Ada Tasks", Proceedings of the Second International Workshop on Real Time Ada Issues, Moreton-Hampstead, Devon, England (June 1988); published in *Ada Letters* VIII, 7 (Fall 1988).

[5] S. Kleiman, "Synchronization in Asynchronous Signal Handling Environments", IEEE POSIX working paper P1003.4-N0299 (May 1991).

[6] P. Santiprabhob, C.S. Chen and T.P. Baker, "Ada Run-Time Kernel: The Implementation", submitted to SERF 91.