PREDICTABLE SYNCHRONIZATION MECHANISMS FOR MULTIPROCESSOR REAL-TIME SYSTEMS.

Lory D. Molesky, Chia Shen and Goran Zlokapa
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

COINS Technical Report 89-106

Predictable Synchronization Mechanisms for Multiprocessor Real-Time Systems.

Lory D. Molesky
Chia Shen
Goran Zlokapa

November 13, 1989

ABSTRACT

Predictability is of paramount concern for hard real-time systems. Every aspect of a real-time system and every primitive provided by the underlying operating system must be bounded and predictable in order to achieve overall predictability. In this paper, we describe a concurrency control synchronization mechanism developed for a next generation multiprocessor real-time kernel, the Spring Kernel. The important features of this mechanism include mutual exclusion with *linear* waiting and *bounded* resource usage. A hardware solution is presented for this problem, which is applicable for not only real-time systems, but also general computing systems. This hardware solution is based on a new synchronization instruction for multiprocessors similar to test-and-set, called *test-and-set-or-branch*.

This work is part of the Spring Project directed by Prof. Krithi Ramamritham and Prof. John A. Stankovic at the University of Massachusetts and is funded in part by the Office of Naval Research under contract N00014-85-K-0398 and by the National Science Foundation under grant DCR-8500332.

1. Introduction

Predictability is of paramount concern for hard real-time systems. The system components that are potentially the most elusive to guarantee predictability are those low level components which are shared by the multiple processors. On a multiprocessor, both shared memory and a shared bus fall into this category. In this paper, we describe the foundations for multiprocessor operating systems support of predictability. We investigate the problems inherent in constructing predictable operating system primitives. In particular, we focus on solutions to the mutual exclusion problem in the domain of real-time systems.

Although the mutual exclusion problem has been extensively studied in a non-real-time context, and many hardware and software solutions exist, real-time systems offer new challenges in dealing with the mutual exclusion issue. In a real-time system, it is not sufficient to ensure only the logical correctness of a task, the timing correctness is equally important. In order to meet the timing constraints of tasks in a real-time system, we must be able to bound the timing of the primitive operations of the operating system. Among the most difficult operating systems primitives to construct with the aim of achieving predictability are those which involve concurrent access to shared data. For example, concurrent interaction between a single scheduler and multiple dispatchers on a multiprocessor may require mutual exclusive access to shared data. Operations for enforcing mutual exclusion operations such as P() and V(), if constructed in a bounded fashion, can provide the framework for other, higher level, bounded operating systems primitives. This boundedness forms a basis for the predictability of the entire system.

The development of solutions for mutual exclusion in real-time multiprocessor operating systems is presented in this paper. We present a solution which improves the *bounded waiting* solution given in [2]. We also provide a hardware solution to it. Current architectures, such as the Motorola 68020, do not provide a single hardware instruction which supports mutual exclusion with bounded waiting. We also discuss the problem of resource usage efficiency in our solution and provide a worst case analysis for both preemptive and non-preemptive real-time runtime environments.

The work presented in this paper is part of the on-going research of the Spring Project. The *Spring Kernel* [15] is currently being built on a VME based 68020 [9] [11] shared memory multiprocessor. Each multiprocessor is composed of up to eight

MVME136A boards. These of the MVME136A boards support features which are typical of shared bus multiprocessors - an asynchronous bus interface, architectural support for *test-and-set* like operations, and a local memory. This memory can either be accessed remotely over the VME bus by (typically) another processor, or locally by the processor which has mapped this local memory. Additional support for multiprocessing is provided through the use of the MPCSR (MultiProcessor Control/Status Registers). One important feature of the MPCSR provides the ability to generate virtual interrupts a selected board, and/or a simultaneous interrupt to multiple boards.

The remainder of this paper is organized as follows. Section 2 discusses background information on multiprocessor synchronization. Existing implementations using test-and-set which provide bounded waiting are discussed in section 3. Section 4 proposes a new, more efficient bounded waiting solution using test-and-set. Section 5 presents a more elaborate mechanism for the implementation of semaphores which reduces shared bus and CPU wastage. Section 6 discusses the use of semaphores supporting bounded waiting in the Spring multiprocessor system. Section 7 concludes the paper.

2. Background on Multiprocessor Synchronization

In this paper, we consider issues of mutual exclusion and synchronization on shared memory multiprocessors. Since the notion of semaphores [3] suffices to provide the underlying support for both mutual exclusion and synchronization, semaphores will be the focus of our discussion.

We are concerned with both efficient solutions for mutual exclusion and those solutions which provide *bounded* waiting. Atomic operations, such as *test-and-set*, are commonly supported on conventional shared memory multiprocessors. This additional hardware support provides for cleaner, more efficient solutions of synchronization primitives [13]. Although pure software solutions for the support of mutual exclusion exist, these solutions achieve correctness at the cost of additional resource usage (memory and bus bandwidth) and therefore are not considered in this paper.

Although the recent technique of wait-free synchronization [7] has potential applicability for real time systems, it is only a *weak* form of synchronization. Wait-free synchronization is *weak* because it cannot support inter-object dependencies. Inter-object dependencies are those which require objects to be ordered based on attributes of these objects. For example, a sorted list is a data structure which contains inter-

object dependencies. Since, for efficiency reasons, the design of the Spring kernel relies heavily on sorted lists, thus wait-free synchronization is not sufficient for our purposes. This is unfortunate, since the worst case analysis of wait-free synchronization techniques is straight forward compared to other synchronization techniques.

Throughout this paper, binary semaphores are used to illustrate access to a critical section. The P() operation acquires the semaphore by atomically setting a shared variable, LOCK. The V() operation releases the semaphore by atomically clearing the lock. C code is used to describe the high level source code, while both pseudo-assembly code and 68020 assembly code is used to describe the low level code.

2.1 Potential Problems with Current Hardware Support

Mutual exclusion in the context of asynchronous parallel processors provides a foundation for many contemporary concurrent computations. Conventional shared memory multiprocessors often support mutual exclusion in the form of atomic *read-modify-write* (RMW) instructions. Systems such as the Motorola MVME136-a, Sequent Symmetry, and the Ultracomputer [11] [12] [6] fall into this category. This support of an atomic RMW instruction is often also referred to as support for test-and-set.

Straightforward use of these hardware implementations however does not meet the requirements of real-time systems because they do not facilitate synchronization with *bounded waiting*. As will be described in detail in the following section, the test-and-set operation is not sufficient to ensure that one processor will not encounter *starvation* when contending for a semaphore. Since hard real-time systems must ensure the predictability of every operation, systems which require concurrent access to shared data must obtain this access in a bounded fashion. This requirement is notably evident in the interaction between multiple dispatchers and the scheduler in the implementation of the Spring Kernel[15].

Today's shared memory multiprocessors' semaphore implementations also suffer from resource wastage [5]. The ubiquitous busy wait loop generates both bus traffic and consumes CPU resources. The bus traffic generated by the busy-wait can be mitigated by a scheme which busy-waits on a cache memory address [12]. Sequent's approach allows each processor only *one* attempt to acquire the semaphore. If this fails, the processor will spin on the cache memory location. In [1], it was noted that this scheme can cause a cascading of cache invalidations, thus causing additional bus

traffic.

2.2 Bounded Bus Access - a Necessary Condition for Bounded Waiting

Hard real-time systems need solutions to the mutual exclusion problem which provide bounded waiting. In a multiprocessor system, unless *bus access* is bounded, no solution can provide a bounded mutual exclusion primitive. Bounded access to a shared bus can, of course, be achieved with the use of a synchronous bus protocol. Synchronous busses are not considered in this paper for a number of reasons, but primarily because their throughput is significantly lower than that of asynchronous busses.

One specific asynchronous bus, the VME bus [11], offers two modes of access, positional (i.e. a daisy-chain) and round robin. The positional scheme favors processors which are electrically closest to the bus arbitration logic. In a positional scheme, the nearest processors can conceivably "hog" the bus while others starve (receive no bus access). When attempting to provide a solution to the mutual exclusion problem which ensures bounded waiting, we cannot configure the bus in a positional mode. The protocol assumed in this paper is thus the round robin protocol.

2.3 Round Robin Mode is not a Sufficient Condition for Bounded Waiting

Round robin mode alone with processors busy-waiting on the semaphore (usually implemented with test-and-set) is however not sufficient to provide bounded waiting. It can be shown that one or more processors can starve when two or more processors contend for a semaphore. It is possible for a subset of the processors to perpetually exchange the lock, starving one or more processors waiting for the lock.

A key issue in the analysis described in this section as well as in other parts of this paper is distinguishing instructions which access the shared bus from instructions which do not access the bus. If all processors involved in contending for a semaphore simultaneously issue an instruction which requires access to the shared bus, these processors can only execute in a round robin fashion. However, a processor in its P() operation can "miss its turn" in the round if it happens to be executing a non-bus master instruction at an inopportune moment in time.

The following example demonstrates the insufficiency of round robin mode alone.

Three processors are perpetually contending for the lock, one of them can starve. Specifically, when all three processors (p_1, p_2, and p_3) are perpetually contending for the lock, a pattern of bus acquisition in the form of $p_1, p_3, p_1, p_3...$ occurs, thus starving p_2.

Suppose three processors, p_1, p_2, and p_3, are involved in the lock acquisition/release sequence. The shared bus is configured in round robin mode such that processors follow each other in a cyclically numerical order (p_1 precedes p_2, p_2 precedes p_3, and p_3 precedes p_1). Further suppose that initially p_1 has the lock (is in its critical section), and p_2 and p_3 are trying to acquire the lock (in P()). Also assume that contention for the resource is sufficiently high such that as soon as a processor performs a V(), it performs another P(). Process p_2 can starve (never get access to the resource) under the following scenario:

P_1 releases the lock by executing a V(). Since, in order to release the lock, p_1 performs a bus operation, it will be p_2's turn to access the bus next. However, if p_2 happens to be executing the branch instruction (refer to the code for P() and V() in figure 2 in section 4.1) when its turn for the bus comes along, p_2 will miss its chance to acquire the lock. Further assume that p_3 does acquire the lock, and after p_3 releases, p_1 acquires the lock. Repeating this sequence, p_2 never acquires the resource, even though it is in P(). This is clearly not a bounded solution.

A similar construction could be presented using only two processors, but the construction with three processors is easier to understand.

We have shown that the round robin bus access mode is a necessary but not a sufficient condition to achieve bounded waiting. The following two sections describe an existing and a new solution, respectively, for semaphore implementations which achieve bounded waiting.

3. A Bounded Implementation Using Test-and-set

The mutual exclusion problem has been studied for a long time, resulting in standard criteria for a correct solution. Any solution to the mutual exclusion problem must meet the correctness criteria relating to *symmetry, process and processor speeds, mutual exclusion*, and *progress*, as noted in [4] and [5]. The symmetry condition disallows the use of a static priority. Assumptions about the process and processors speeds are

```
boolean LOCK;
P()                                          V()
{                                            {
    while !(test-and-set(LOCK)) ;                LOCK = false;
}                                            }
```

Figure 1: The Basic Busy-wait Loop

not allowed. The mutual exclusion condition allows only one process to be executing in its critical section at any point in time. The progress condition ensures that, if a process requests to enter a critical section which is not in use, it will be allowed to eventually enter the critical section. In the context of operating systems for real-time systems, this eventuality does not suffice. What is needed is the guarantee of bounded waiting. *Bounded waiting* and *linear waiting* are defined as follows [2]:

- **Definition:**
  Bounded waiting is achieved if there is a constant k such that if a process is in its busy-wait loop, then that process will enter its critical region before any other process has entered its critical region more than k times. When k=1, this property is called *linear waiting.*

The simplest form of acquisition and release procedures for a semaphore, P() and V(), is shown in figure 1. It is well known that this simple implementation with test-and-set satisfies the *symmetry, processor speeds, mutual exclusion*, and *progress* conditions. This implementation of P() and V() does not however satisfy the *bounded* waiting condition. The problem, as mentioned in the previous section, arises when one process can starve when two or more processes are involved in the contention.

In [2], Burns presents a mutual exclusion solution for shared memory multiprocessors which does achieve bounded waiting. This solution augments the test-and-set instruction and the single shared memory lock address with N additional binary shared variables. N corresponds to the maximum number of processors involved in contention for the critical section. Once a processor fails on the test-and-set in its P() region, it asserts the appropriate flag in the waiting array. When a release of the semaphore occurs in the V() section, the next processor (in some predefined cyclic order) with its flag set in the waiting array is allowed to acquire the semaphore.

To prove that an implementation achieves linear waiting, all that is necessary is to demonstrate a cyclic ordering of waiting processes. Since the waiting array is scanned in cyclic order, (e.g. from 0, 1 ... N -1 back to 0), if processor p is waiting (e.g., has entered P()), it will enter its critical section within N - 1 turns.

4. Efficient Bounded Semaphores

This section offers a more efficient solution to the bounded mutual exclusion problem than the one presented by Burns [2]. In real-time systems, more often than not, scheduling decisions are made based on execution times of tasks. This demands knowledge about instruction timing properties. Our solution exploits this knowledge to obtain an upper bound on the wait for the P() operation. Unlike the solution provided by Burns, the new solution needs no additional shared memory locations (i.e. the waiting array can be dispensed with). This solution is based on test-and-set, and strongly resembles the non-bounded solution in terms of efficiency of code and space. Using this bounded busy-wait construct, more elaborate bounded mutual exclusion solutions are constructed in section 5 which provide predictability, and minimize both CPU wastage and bus wastage.

It should be noted that even though we reason about the instruction timing properties in our solution presented below, this does not violate the *process and processor speeds* condition in the correctness criteria for mutual exclusion. The instruction timing properties concern the *absolute* time some instruction takes, not the *speed* of the processor or the *pace* of some process.

4.1 Our Solution

In section 3 it was shown that a round robin bus protocol alone was not sufficient for a bounded mutual exclusion protocol. If it can be demonstrated that a particular semaphore implementation enforces a cyclic ordering of the waiting processors, then the implementation is bounded. Moreover, as Burns illustrated, this implementation is linear. In order to prove this cyclic ordering of waiting processors, we reason about the possible events after the processor which holds the lock releases it. In the following discussion, it is assumed that the round robin protocol grants bus access to processors in numerical order (that is p_{i+1} follows p_i).

A few details pertaining to the general shared bus arbitration are necessary before the new protocol can be presented.

- **Definition:**
  Only one processor is allowed to control the shared bus at any point in time, this processor is called the *bus master*; other processors are called *non-bus masters.*

- **Definition:**
  If a processor p_i initiates a bus request which cannot be satisfied because another processor is the bus master, then the bus instruction issued by p_i becomes *pending.* In the context of bus operation in a round robin mode, a pending bus instruction is essentially queued by the hardware; the enqueueing order is predetermined, as defined by the round robin protocol.

The basic approach in the construction of an implementation which achieves linear waiting is to design the V() operation such that the release of the semaphore holds the bus long enough to ensure that the closest processor in the round robin cycle waiting (i.e. in its P() section) will be guaranteed to execute its test-and-set operation when its "round" is active. Thus, by ensuring that the non-bus master component of the acquisition loop of P() is as small as the bus master time of the atomic release instruction in V(), the cyclic waiting order can be ensured.

For the purposes of this discussion, we assume that processes waiting in the P() operation are non-preemptable. In a preemptable environment, one could argue that starvation could occur under degenerate conditions by an inopportune preemption of a particular process immediately prior to the acquisition of the semaphore. In other words, using an adversary argument, a process p_i will starve if, each time the bus mastership is about to be granted to the processor executing p_i, p_i is preempted. Additionally, we assume that a process cannot be preempted while in its critical section. If this were to occur, all other processes could wait indefinitely. An extended discussion of issues involved in preemptive scheduling follows in section 5.

A generalized form of a the P() and V() operations is presented in figure 2. When the semaphore is in use, the semaphore's state will be *set.* Otherwise the semaphore is available for acquisition, and is referred to as *clear.* The P() operation consists of both instructions which access the shared bus, and which do not access the shared bus. We assume that at least one instruction in the busy-wait loop of P() is a bus

```
P()                                    V()
*                                      *
* Address of lock is in A0             * Address of lock is in A0
*                                      *

SPIN:
      test-and-set          (A0)               clear (A0)
      conditional-branch    SPIN
```

Figure 2: Generalized P() and V() Routines

master instruction. The V() operation consists of at least one instruction which is a bus master instruction. This instruction, `clear`, performs the actual "`clear`ing" of the semaphore. In the following theorem, the event termed *releasing* the semaphore refers to the point in time when the processor executing the `clear` instruction (in V()) transfers its state from being the bus master to non-bus master. At *release* time, it is known that the semaphore is cleared.

Deferred Bus Theorem:

> If the total worst case non-bus master time of the busy-wait loop (in P()) is less than the best case bus master time of the release instruction, and if processor p_j is the closest processor (in the round robin ordering) busy-waiting for semaphore s when processor p_i releases s (in V()), then p_j will be the next processor to acquire s.

Proof:

To prove the theorem, the two possible circumstances which occur when p_i releases the semaphore are enumerated. These correspond to the two instructions of the busy-wait loop of the P() operation of p_j. Either p_j is executing the bus master instruction `test-and-set` (case 1), or it is executing the non-bus master instruction `conditional-branch` (case 2).

1. **A `test-and-set` was pending on p_j when a `clear` by p_i was executed.** Since the `test-and-set` is pending and p_j is the next processor waiting, p_j acquires the semaphore next.

2. **A `test-and-set` was not pending on p_j when a `clear` by p_i was executed.** Since the worst case duration of the non-bus master time of the busy-wait acquisition loop is less than the bus master time of the release instruction, the `test-and-set` issued by p_j will be invoked before the `clear` by p_i is completed. Thus the `test-and-set` of p_j will become pending before the clear by p_i completes. By case 1, this implies that p_i acquires the semaphore next.

Since in either case, p_j acquires the semaphore next, the proof is complete.

Q.E.D.

4.2 A Hardware Solution

This section presents a hardware solution for a semaphore which provides bounded waiting. In addition to being applicable to real-time computing systems, this hardware solution is also applicable to general computing systems. This general applicability is achieved by eliminating the need to know instruction execution times of the P() and V() operations.

Recall that the basic problem in achieving predictability in the P() and V() routines is that the next processor waiting in the round robin ordering could be executing its `conditional-branch` instruction when its "turn" for the bus arrives. The Deferred Bus theorem ascertained that the turn would not actually be missed under certain instruction execution assumptions. The underlying problem here is that the `conditional-branch` is a non-bus master instruction.

- **Definition:**
  The `test-and-set-or-branch` instruction first locks the bus, then tests the operand specified by the effective address. The remaining steps are conditional on the value of the operand. If the operand is:
  - **zero:**
    The operand is set to one, the bus is released, and control is returned.
  - **non-zero:**
    Simultaneously, the bus is first released, then a new bus request is initiated in order to reexecute the `test-and-set-or-branch`.

By combining the `conditional-branch` instruction with the `test-and-set` in-

```
P()
*
* Address of lock is in A0.
*
*

SPIN:
   test-and-set-or-branch (A0)
```

```
V()
*
* Address of lock is in A0.
* clear is a
* bus-master instruction.
*

   clear (A0)
```

Figure 3: Generalized P() and V() Routines using `test-and-set-or-branch`

struction into one bus master instruction (`test-and-set-or-branch`), we can eliminate all assumptions about instruction execution time and still support a semaphore which provides bounded waiting. `test-and-set-or-branch`, like `test-and-set`, is a bus master instruction, locking the bus until the entire instruction has completed. Note that, after an unsuccessful `test-and-set-or-branch` (the operand was non-zero), control of the bus is released. The round robin bus grant mode will prevent the releasing processor from hogging the bus - the releasing processor will only acquire the bus two or more successive times if no other processor is waiting.

The non-bus master time of the busy-wait loop will be zero if a careful implementation of the release/request sequence of the `test-and-set-or-branch` is constructed. Whenever the test portion of the `test-and-set-or-branch` of processor p_i fails, bus arbitration is initiated while still keeping a request for p_i pending.

The implementation of test-and-set-or-branch can be efficient. Depending on the hardware/firmware implementation, this combined instruction may not necessarily hold the bus longer than the standard `test-and-set` instruction.

The new specialized P() and V() operations are shown in figure 3. This implementation meets the requirements of the Deferred Bus theorem. i.e. the worst case non-bus master time of the instructions in P() is zero, thus it provides bounded (linear) waiting. Since the worst case non-bus master time is zero, there is no need to compare instruction execution times between the P() and V() operations - as long as the `clear` instruction in V() is a bus master operation, the Deferred Bus theorem is true.

4.3 Extending Current Architectures

Current architectures, such as the Motorola 68020/68851 CPU and paged memory management chip set, do not provide support for the Deferred Bus theorem. It is not possible to construct P() and V() operations such that the total worst case non-bus master time of the busy-wait loop in P() is less then the best case bus master time of the release instruction of V(). This will be demonstrated primarily by examining the timing properties of the RMW type instructions available for this chip set. Extensions specific to this chip set which support the Deferred Bus theorem are presented in the following discussion.

In the 68020/68851 architecture, the maximum time the shared bus is locked during a RMW operation is 8 cycles [10]. The worst case execution time of a conditional branch instruction is 9 cycles. These figures alone are enough to demonstrate that the Deferred Bus theorem cannot hold for this architecture - the release instruction can hold the bus for at most 8 cycles, which cannot be guaranteed to be longer than the non-bus master time of the busy-wait loop (since the busy-wait loop contains a conditional branch).

A slight divergence into the policies and mechanisms adopted in the Motorola 68020/68851 architecture is necessary before specific recommendations are made for extending this architecture. In the Motorola 68020/68851 architecture, the 68851 (the PMMU) not only handles address translations (from virtual to physical memory), but also performs bus requesting. The CPU (68020) supplies the PMMU with instructions and operand addresses over dedicated bus lines.

The support for limiting the maximum time that the shared bus is locked during a RMW to a small number of cycles is rather interesting, involving interaction with another functional unit of the PMMU, the ATC (Address Translation Cache). When a TAS (test-and-set) operation is detected by the PMMU, the first step taken by the PMMU is to become bus master and try to assert the RMC signal (Read Modify Cycle) on the shared bus. The next step is to translate the virtual address to a physical address. If an ATC miss occurs on this address, both the RMC and the TAS operation are aborted. The PMMU instructs the CPU to reexecute the TAS instruction. This policy is used to reduce the maximum amount of time that the shared bus is locked by a processor – address translations are very expensive (40 - 68) cycles.

Extending the 68020/68851 architecture to support a `test-and-set-or-branch` (TASOB) instruction will require similar interaction between the CPU and PMMU.

Once the CPU has communicated the TASOB instruction and appropriate operands to the PMMU and the physical address has been cached, the PMMU need not return control to the CPU until the semaphore has been acquired. We effectively implement a busy-wait loop inside the PMMU. Until a successful acquisition of the semaphore, the PMMU successively contends for the shared bus, locks the shared bus, attempts acquisition of the semaphore, then unlocks the bus.

The non-bus master time of the TASOB will be limited to the time required to communicate the opcode and operands from the CPU to the PMMU, plus the potential cost of performing an address translation. Moreover, since these costs occur only *once* per P() operation, they can be easily accounted for by adding a constant to the overhead of the P() operation. Since these non-bus master times occur at the beginning of the busy-wait loop, they are not part of the busy-wait loop and thus are discounted from violating the Deferred Bus theorem. As discussed in the previous subsection, the implementor of V() must use a bus-master instruction to release the semaphore.

4.4 Bound on Bus Usage

Although our solution is more efficient than that of Burns, it nevertheless wastes bus and CPU resources. An upper bound on the shared bus wastage is presented in this section, while solutions which mitigate the CPU and bus wastage are presented in the next section. The notation *T()* indicates the exact number of steps required to execute a program fragment.

We have shown that linear bounded access to the critical section will be granted in a cyclic fashion. Thus, any processor performing the P() operation will wait for at most $N-1$ other processors before being granted access to the critical section. Each of the $N-1$ other processors will cause a delay of $T(P()) + T(criticalsection) + T(V())$. During this cumulative delay $((N-1) * (T(P()) + T(criticalsection) + T(V())))$, the acquiring processor will be performing bus requests. For long critical sections, this bus resource wastage should be avoided. The next section discusses solutions which minimize this bus wastage.

5. Reducing Bus Traffic with Queued Semaphores

Section 4 presented an efficient implementation of linearly bounded semaphores. In this section, we extend the busy-wait implementation to minimize bus traffic and CPU wastage. The linearly bounded semaphores are used as a basis for the extension presented in this section. Bus traffic is minimized by enqueuing any waiting process in a shared queue. This technique is similar to one used by monitors [8], but is instead applied to semaphores.

The queue is FIFO, constructed by allocating a contiguous area of memory and maintaining a pointer to the current free slot. Enough slots must be allocated to accommodate the number of processors (for non-preemptive scheduling) or the number of processes (for preemptive scheduling).

A queued semaphore s_q is defined as a semaphore which serializes access to the queue. Once the semaphore has been acquired by process (processor) p, p may perform the enqueue of its identifier. The technique described in section 4 is used to bound the P() operation, thus bounding the enqueue operation by bounding the time from the initial request to enqueue until the actual start of the enqueue. The V() operation performs a dequeue of the p identifier. As in the P() operation, the semaphore s_q must be acquired in order to perform the dequeue operation. As noted, the semaphore acquisition is bounded.

The method of waiting to be dequeued and being dequeued is different in the preemptive and non-preemptive runtime environments. The distinction and motivation of the alternatives are discussed in the following subsections.

5.1 Non-Preemptive Runtime Environment

In a non-preemptive runtime environment, no attempt is made to utilize the CPU cycles of a processor waiting for a semaphore. Thus, eliminating CPU wastage during the P() operation on a semaphore is of little concern. This fact is exploited in the non-preemptive solution with a busy-wait on a local memory address.

After the first attempt to acquire the semaphore fails, instead of continuing to busy-wait on a shared memory location, the acquiring processor first enqueues itself on a shared queue, then spins on a local memory location. This action eliminates VME bus traffic (there is a separate bus from the CPU to local memory). When the processor

which is currently using the resource releases it (the V() operation), it wakes up the next waiting process in the queue by clearing the appropriate memory location (local to the waiting processor).

In the solution for non-preemptive runtime environment, the strong semaphore implementation requires at most one VME bus cycle to fail to acquire the lock, and will wait at most N - 1 cycles to enqueue itself. An additional VME bus cycle is required to release the semaphore to a waiting processor. Thus we have achieved an T(N) bound on the number of VME bus requests per semaphore request. This is a significant improvement on the bound for the busy-waiting technique which was a function of the size of the critical section.

5.2 Preemptive Runtime Environment

In a preemptive runtime environment, a processor may preempt a process which has failed to acquire the semaphore on the first try. The entire scenario is similar to that of a non-preemptive environment, except that after the enqueue operations, the current process will preempt itself and another process may continue to use the CPU. The notification process occurring during the dequeue is more complicated in the preemptive runtime environment, and requires interaction with the operating system. Explicit notification of the preempted process will minimally require the processor identifier and the process identifier for naming purposes.

In order to preserve the bounded nature of accessing a resource, a couple of assumptions were stated earlier about where preemption could (not) occur. A process is not allowed to be preempted while in its critical section, or while in its P() operation.

6. Use of Bounded Waiting Semaphores

In this section, areas of hard real-time systems which can exploit bounded waiting semaphores are discussed. The discussion focuses on the design of the Spring kernel [15] which supports the notion of an on-line guarantee for *dynamic* task arrivals. The Spring architecture is composed of a collection of multiprocessors on a distributed network. In the following discussion, we focus on concurrency involved on a single multiprocessor.

In the Spring approach, application tasks are scheduled such that resource conflicts

are avoided [14]. However, a multiprocessor operating system supporting concurrent execution of tasks does require support for mutual exclusion. Since, to achieve predictability, the overhead of the operating system must be accounted for in the worst case computation time of application tasks, all operating systems operations must also be bounded. For example, since an application task's worst case computation time must also include its dispatch time, the dispatch time (an operating system primitive) must be bounded.

Generally, one Spring node has one system processor and multiple application processors. The scheduler is located on the system processor, while a dispatcher runs on each application processor. Efficient system support for the on-line guarantee routine centers around concurrent activity of the scheduler and the multiple dispatchers. The primary data structure shared by the scheduler and multiple dispatchers is the system task table (STT). In order to facilitate concurrent access to the STT by the dispatchers, the STT is partitioned (with linked lists) based on the application processor to which each task is assigned. Concurrent access to the STT by the scheduler and dispatcher processes is required under many circumstances. Since concurrent access to shared data is required by the scheduler and dispatcher, and these costs contribute to an application task's worst case computation time, this concurrent access must also be bounded.

Another area where bounded semaphores are useful in a predictable hard real-time system is for enforcing mutual exclusive access to resources for certain kinds of application tasks. If a tasks' access patterns to a resource are of long duration and/or are not very frequent, techniques avoiding resource conflicts (e.g., via resource segmentation and partitioning with an integrated CPU scheduling with resource allocation algorithm) can be used. However, an alternative approach can be taken in cases where access to resources is frequent and/or of very short duration. In particular, consider a pair of application processes which require exclusive access to a shared data area frequently, and access to this shared data is of limited duration. Segmenting these tasks into one task per resource request is not practical, especially if the duration of the task is less than the overhead of the scheduler. In these situations of small granularity resource access, the technique of using a bounded semaphore is much more realistic. If the interleaved access to shared data is included in the worst case computation time of each task, tasks requiring exclusive access to identical resources may thus be scheduled to execute concurrently.

7. Conclusion

Techniques for avoiding shared bus contention in the solution to mutual exclusion problems have been discussed in this paper. A queued semaphore technique for hard real-time systems has been presented which requires at most a linear number of bus cycles to acquire the semaphore. This implementation is based on an efficient implementation of semaphores with linear waiting.

The efficient implementation of semaphores with linear waiting is achieved by reasoning about the class of machine instruction timing properties. Specifically, assuming a round robin bus protocol, it was shown that if the bus-master time of the release instruction of V() is at least as long as the worst case non bus-master components of the busy-wait loop of P(), then the semaphore implementation provides linear waiting. Conversely, if one is not careful and uses an implementation where this is not true, then unbounded waiting can occur. Bounded access to concurrent data is one essential component in providing predictability for hard real-time systems such as the Spring kernel.

A hardware solution for semaphores providing linear waiting was presented which is superior to the solution of Burns in both space and time. Burns' solution requires an extra boolean variable per processor. His solution has more instructions in both the P() and V() operations. In particular, our busy wait loop is tighter (less instructions) and is thus more responsive. Our hardware solution, based on a new `test-and-set-or-branch` instruction, is applicable to general computing systems, including real-time systems.

Extensions to this work involve a further reduction of bus traffic. Although the queued semaphore solution presented in this paper effectively reduces the shared bus traffic on a multiprocessor, the amount of bus traffic it requires is not *minimum*. We have defined an *asymmetric protocol* for implementing semaphores which in fact *minimizes* bus access. This work will be the subject of a future paper.

8. Acknowledgements

The authors of this paper wish to thank Professor Krithi Ramamritham, Professor John A. Stankovic, and Victor Yodaiken for their insightful discussion of some of the ideas presented in this paper.

References

[1] T. E. Anderson, E. D. Lazowska, and H. M. Levy. The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors. Technical Report 88–09–04, University of Washington, September 1988.

[2] J. E. Burns. Mutual Exclusion with Linear Waiting using Binary Shared Variables. *SIGACT News*, 10(2), Summer 1978.

[3] E. W. Dijkstra. The Structure of the "THE"-Multiprogramming System. *Communications of the ACM*, 11(5), May 1968.

[4] E. W. Dijkstra. Hierarchical Ordering of Sequential Processes. *Acta Informatica*, 1, 1971.

[5] A. Dinning. A Survey of Synchronization Methods for Parallel Computers. *Computer*, 22(7), July 1989.

[6] A Gottlieb et. al. The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer. *IEEE Transactions on Computers*, c-32(2), February 1983.

[7] M. P. Herlihy. Impossibility and Universality Results for Wait-Free Synchronization. Technical Report 88–140, Carnegie Mellon, May 1988.

[8] C. Hoare. Monitors: An Operating System Structuring Concept. *CACM*, 17(10), October 1974.

[9] Motorola Inc. *MC68020 32-Bit Microprocessor User's Manual.* Prentice-hall, Englewood Cliffs, N.J., 1985.

[10] Motorola Inc. *MC68851 Paged Memory Management Unit User's Manual.* Prentice-hall, Englewood Cliffs, N.J., 1986.

[11] Motorola Inc. *MVME135, MVME135-1, MVME135A, MVME136, and MVME136A 32-Bit Microcomputers User's Manual.* Motorola Inc., 1989.

[12] Sequent Computer Systems Inc. *Sequent Symmetry Technical Summary.* Sequent Computer Systems, Inc., 1988.

[13] J. L. Peterson and A. Silberschatz. *Operating System Concepts.* Addison-Wesley, Reading, Massachusetts, 1985.

[14] K. Ramamritham, J. A. Stankovic, and P. Shiah. O(n) Scheduling Algorithms for Real-Time Multiprocessor Systems. In *the 9th International Conference on Parallel Processing*, June 1989.

[15] J. A. Stankovic and K. Ramamritham. The Spring Kernel: A New Paradigm for Real-time Operating Systems. *Operating Systems Review*, 23(3), July 1989.