A Simple Proof Technique for Priority-scheduled Systems

James H. Anderson [1]

The University of North Carolina, Chapel Hill, NC 27599

Mark Moir [2]

Sun Microsystems, Inc., Burlington, MA 01803

Srikanth Ramamurthy

IBM-Transarc Labs, Pittsburgh, PA 15222

Abstract

A simple proof method is presented for proving invariance properties of concurrent programs in priority-scheduled systems. This method is illustrated by using it to establish the correctness of a simple wait-free consensus algorithm for priority-scheduled uniprocessor systems. This consensus algorithm is of interest in its own right because is shows that atomic read and write operations are universal in priority-scheduled uniprocessor systems, *i.e.*, they can be used to implement *any* shared object in such a system in a wait-free manner. This stands in contrast to fully asynchronous systems, where strong synchronization primitives such as compare-and-swap are needed for universality.

Key words: Consensus, multiprogramming, program correctness, real-time systems, scheduling, wait-freedom

1 Introduction

In this paper, we introduce a simple proof method for proving invariance properties of concurrent programs in priority-scheduled systems. The key idea

[1] Supported, in part, by NSF grant CCR-9732916.
[2] Supported, in part, by an NSF CAREER award, grant number CCR-9702767.

behind this proof technique is to modify the standard definition of an "enabled statement" to reflect the fact that a process does not take steps while a higher-priority process on its processor is enabled to take steps.

As a case study, we consider the use of wait-free shared objects in priority-scheduled systems. A shared object implementation is *wait-free* if and only if it satisfies the following notion of delay tolerance: if several processes perform object invocations concurrently, and if some proper subset of these processes stop taking steps, then each of the remaining processes completes its object invocation in a finite number of its own steps.

Herlihy has shown that, in general, strong synchronization primitives such as compare-and-swap (CAS) are necessary for wait-free implementations [9]. This result follows from a ranking of shared objects by "consensus number". An object has *consensus number* N if it can be used to implement wait-free consensus for N processes, but not for $N + 1$ processes. Herlihy showed that synchronization objects with unbounded consensus numbers are necessary in general-purpose wait-free object implementations. Such objects are called *universal* because they can be used to implement any other object. An example of a universal object is a shared variable accessible by read and CAS operations.

In this paper, we present a wait-free consensus algorithm for multiprogrammed uniprocessor systems in which processes are scheduled by priority. This algorithm uses only atomic read and write operations and is correct for any number of processes. It follows that reads and writes are universal in priority-scheduled uniprocessor systems. The distinguishing characteristic of such a system is that process interleavings are restricted — if process p can preempt process q, then q cannot preempt p. Our consensus algorithm heavily exploits this fact.

Real-time systems are an important class of systems in which priority scheduling is usually used. In most real-time systems, shared objects are implemented using lock-based mechanisms. Unfortunately, the use of such mechanisms may lead to priority inversions. A *priority inversion* is said to exist when a process blocks on a process of lower priority. Conventional mechanisms for dealing with priority inversions [11,13] rely on the kernel to dynamically raise the priority of a process causing a priority inversion so that the duration of the priority inversion is bounded. This adds complexity to the kernel and makes dynamic process creation and removal more difficult to support. Wait-free shared objects cannot cause priority inversions, and thus have an advantage over lock-based mechanisms. The results of this paper imply that in uniprocessor real-time systems that are priority scheduled, *any* shared object can be implemented in a wait-free manner without strong synchronization primitives.

The rest of this paper is organized as follows. In Sec. 2, we present our consensus algorithm. Then, in Sec. 3, we present our proof method, and use it to

prove that the algorithm is correct. Concluding remarks appear in Sec. 4.

2 Consensus Algorithm

In the consensus problem, each process p has two private variables $p.input$ and $p.output$. Initially, $p.output = \perp$ and $p.input \neq \perp$; we will assume that $p.input$ ranges over $0..C$, where $C \geq 1$. $p.output$ is updated exactly once by p, while the value of $p.input$ remains unchanged. The value of $p.input$ is called p's *input value*, and the value assigned to $p.output$ is called p's *output value*. Upon termination of all processes, the following condition must hold.

$$(\exists q :: (\forall p :: p.output = q.input))$$

In addition, each process is required to be wait-free; because each process may potentially finish the algorithm before any other process begins, this requirement eliminates trivial solutions in which each process chooses a preselected input value as its output value. Loui and Abu-Amara have proven the following theorem regarding consensus in asynchronous shared-memory systems [10].

Theorem 1 (Loui and Abu-Amara) *The consensus problem has no solution (in an asynchronous system) using only atomic read and write operations.*

In this section, we show that the consensus problem *can* be solved using only atomic reads and writes in a priority-scheduled uniprocessor system. As a first step towards a solution, consider the following simple (incorrect) algorithm.

```
shared variable Final: 0..C ∪ ⊥
initially Final = ⊥

process p:                    /* 0 ≤ p < N */
1: if Final = ⊥ then
2:     Final := p.input
   fi;
3: p.output := Final
```

In this algorithm, the final decision value is recorded in the shared variable $Final$. Process p assigns its input value to $Final$ if it finds that $Final = \perp$ holds. Unfortunately, this simple algorithm does not work. Consider two processes p and q such that $p.input = 5$ and $q.input = 7$, where q has higher priority. The following sequence of statement executions is possible.

- p executes statement 1 and reads $Final = \perp$.
- q preempts p, executes statement 1, and reads $Final = \perp$.

```
shared variable Propose, Final: 0..C ∪ ⊥
initially Propose = ⊥ ∧ Final = ⊥

process p:                          /* 0 ≤ p < N */
private variable p.temp, p.input, p.output: 0..C ∪ ⊥

1: if Propose = ⊥ then
2:     Propose := p.input
   fi;
3: if Final = ⊥ then
4:     p.temp := Propose;
5:     Final := p.temp
   fi;
6: p.output := Final
```

Fig. 1. Consensus using reads and writes.

- q executes statements 2 and 3, establishing $Final = 7$ and $q.output = 7$.
- p executes statements 2 and 3, establishing $Final = 5$ and $q.output = 5$.

This counterexample suggests that at least two shared variables are needed, because any process that is preempted when it is enabled to write a shared variable will immediately overwrite the previous contents of that variable when it resumes execution. We call such a write a "late write".

Our next attempt is an algorithm in which each process tries to write two shared variables in sequence. This algorithm is shown in Fig. 1. This algorithm is correct, as we shall prove in the next section. A key property of the algorithm is that only the first value written to *Propose* can be written to *Final*. If process p is preempted before executing statement 2 (*i.e.*, it is enabled to perform a late write of *Propose*), then when it resumes execution, $Final \neq \bot$ holds, and hence statements 4 and 5 are not executed by p. This is because any process that preempts p runs to completion before relinquishing the processor. If process p is preempted before executing statement 5 (*i.e.*, it is enabled to perform a late write of *Final*), then when it resumes execution, its overwrite of *Final* will leave the value of *Final* unchanged. This is because only the first-written value to *Propose* can be written to *Final*, and *Propose* has already been written prior to p's preemption.

3 Correctness Proof

We prove that the program in Fig. 1 satisfies the requirements of the consensus problem by establishing several invariants. Before presenting these invariants, we first state some notational conventions that will be used hereafter, and discuss our proof obligations in establishing the required invariants.

Notational Conventions: Unless otherwise specified, we assume that p and q range over $\{0, \ldots, N-1\}$. If p and q appear as free variables in an assertion, then they are assumed to be universally quantified. We assume that processes are ordered by priority, *i.e.*, process p has higher priority than process q if and only if $p < q$.

Each numbered statement is assumed to be atomic. (Observe that each such statement accesses at most one shared variable.) Statement number k of process p is denoted $k.p$.

Let S be a subset of the statement labels in process p. Then, $p@S$ holds if and only if the program counter for process p equals some value in S.

For each process p, we assume that $p@\{0\}$ holds initially, and that statement 6 establishes $p@\{7\}$. Each process's execution is controlled by a "scheduler" process. If $p@\{0\}$ holds, then the scheduler may establish $p@\{1\}$. The scheduler may not modify any variables or program counters in any other way.

We say that a process p is *active* if and only if it has begun executing the algorithm, but has not yet terminated. For our algorithm, this is formalized by $active(p) \equiv \neg p@\{0, 7\}$. We say that a process p is *running* if and only if it has the highest priority among all active processes. Formally, $running(p)$ is true if and only if $active(p) \ \wedge \ (\forall q : q < p :: \neg active(q))$ holds. We let $Enabled(k.p)$ be a predicate that is true if and only if $p@\{k\} \ \wedge \ running(p)$ holds.

All of the invariants given below are implications. To avoid excessive parentheses, we will assume that "$\Rightarrow$" has the lowest binding power of any symbol. $\Box$

Proof Obligations: An assertion is an invariant if it is initially true and stable. An assertion P is *stable* for a program if it cannot be falsified by any statement execution of that program. Formally, for each statement $k.p$, $P \ \wedge \ Enabled(k.p) \ \Rightarrow \ wp(k.p, \ P)$ holds, where wp is the "weakest precondition" predicate transformer [8]. Other previously-established invariants may be used when proving stability. $\Box$

Note that our notion of stability differs from that in an asynchronous system only in how $Enabled(k.p)$ is defined. In an asynchronous system, $Enabled(k.p)$ would be defined to be simply $p@\{k\}$.

We now prove that the program in Fig. 1 is correct by establishing six invariants. The first three invariants are quite straightforward, and thus are stated without proof. For the remaining invariants, a proof is given.

The first two invariants follow easily from the code of process p and the fact

that no process assigns the value $\perp$ to either *Propose* or *Final*.

invariant $p@\{3..7\} \;\Rightarrow\; \mathit{Propose} \neq \perp$ (I0)

invariant $p@\{6,7\} \;\Rightarrow\; \mathit{Final} \neq \perp$ (I1)

The next invariant holds because each value assigned to *Propose* and hence *Final* is the input value of some process.

invariant $\mathit{Final} \neq \perp \;\Rightarrow\; (\exists r :: \mathit{Final} = r.\mathit{input})$ (I2)

According to the next invariant, (I3), if $p@\{3..6\}$ holds, and if q has higher priority than p, then $q@\{2\}$ is false.

invariant $p@\{3..6\} \;\Rightarrow\; (\forall q : q < p : \neg q@\{2\})$ (I3)

Proof: Initially, $p@\{0\}$ holds, so (I3) holds. (I3) can be potentially falsified only by statements $1.q$ and $2.p$, where q is any process (only statements $1.p$ and $2.p$ can establish $p@\{3..6\}$, and only statement $1.q$ establishes $q@\{2\}$). By the definition of $\mathit{Enabled}(2.p)$, statement $2.p$ can establish $p@\{3..6\}$ only if $(\forall q : q < p : \neg q@\{2\})$ holds. Hence, it cannot falsify (I3). For $1.q$, we have the following.

$$\begin{array}{ll}
& \text{I3} \;\wedge\; \mathit{Enabled}(1.q) \\
\Rightarrow & (\mathit{Enabled}(1.q) \;\wedge\; \neg p@\{3..6\}) \;\vee \\
& (\mathit{Enabled}(1.q) \;\wedge\; p@\{3..6\} \;\wedge\; p \leq q) \;\vee \\
& (\mathit{Enabled}(1.q) \;\wedge\; p@\{3..6\} \;\wedge\; p > q) \\
& \qquad\qquad \text{, predicate calculus.} \\
\Rightarrow & (\mathit{Enabled}(1.q) \;\wedge\; \neg p@\{3..6\}) \;\vee \\
& (\mathit{Enabled}(1.q) \;\wedge\; p@\{3..6\} \;\wedge\; p > q) \\
& \qquad\qquad \text{, by the definition of } \mathit{Enabled}(1.q), \text{ the} \\
& \qquad\qquad \text{second disjunct above is false.} \\
\Rightarrow & (\mathit{Enabled}(1.q) \;\wedge\; \neg p@\{3..6\}) \;\vee \\
& (\mathit{Enabled}(1.q) \;\wedge\; p@\{3..6\} \;\wedge\; p > q \;\wedge \mathit{Propose} \neq \perp) \\
& \qquad\qquad \text{, by (I0).} \\
\Rightarrow & \mathit{wp}(1.q,\ \text{I3}) \qquad\qquad \text{, by the definition of (I3) and the} \\
& \qquad\qquad \text{monotonicity of } \mathit{wp}. \quad \square
\end{array}$$

According to (I4), stated next, the execution of statement 5 has the effect of

copying the current value of *Propose* to *Final*.

invariant $p@\{5\} \;\Rightarrow\; p.temp = Propose$ (I4)

Proof: $p@\{5\}$ is initially false, so (I4) holds initially. (I4) can be potentially falsified only by statements $4.p$ (which establishes the antecedent and updates $p.temp$) and $2.q$ (which updates *Propose*), where q is any process. However, statement $4.p$ establishes the consequent of (I4). Hence, it cannot falsify (I4). For statement $2.q$, we have the following.

I4 $\wedge$ *Enabled*$(2.q)$

$\Rightarrow$ $(p@\{5\} \wedge Enabled(2.q)) \vee (\neg p@\{5\})$
, predicate calculus.

$\Rightarrow$ $((\forall r < p : \neg r@\{2\}) \wedge q < p \wedge q@\{2\}) \vee (\neg p@\{5\})$
, by (I3) and the definition of *Enabled*$(2.q)$.

$\Rightarrow$ $\neg p@\{5\}$, the first disjunct above is false.

$\Rightarrow$ $wp(2.q,$ I4$)$, by the definition of (I4) and the monotonicity of wp. $\square$

According to the next invariant, (I5), if $p@\{4,5\}$ holds, then either *Final* hasn't yet been updated, or $Final = Propose$ has been established.

invariant $p@\{4,5\} \;\Rightarrow\; Final = \bot \;\vee\; Final = Propose$ (I5)

Proof: $p@\{4,5\}$ is initially false, so (I5) holds initially. (I5) can be potentially falsified only by statements $3.p$ (which may establish $p@\{4,5\}$), $2.q$ (which updates *Propose*), and $5.q$ (which updates *Final*), where q is any process. However, statement $3.p$ does not modify *Final* and establishes $p@\{4,5\}$ only if $Final = \bot$ holds. Hence, it cannot falsify (I5). By (I3) and the definition of *Enabled*$(2.q)$, if *Enabled*$(2.q)$ holds, then $p@\{4,5\}$ is false. Hence, statement $2.q$ cannot falsify (I5). For statement $5.q$, we have the following.

I5 $\wedge$ *Enabled*$(5.q)$

$\Rightarrow$ $(p@\{4,5\} \wedge Enabled(5.q)) \vee (\neg p@\{4,5\})$
, predicate calculus.

$\Rightarrow$ $(p@\{4,5\} \wedge Enabled(5.q) \wedge Propose = q.temp) \vee (\neg p@\{4,5\})$
, by (I4) and the definition of *Enabled*$(5.q)$.

$\Rightarrow$ $wp(5.q,\ \text{I5})$, by the definition of (I5) and the monotonicity of wp. $\Box$

According to the final invariant, (I6), the output value of each process is the (unique) input value copied to *Final*.

invariant $p@\{7\} \ \Rightarrow \ p.output = Final$ (I6)

Proof: Initially, $p@\{7\}$ is false, and hence (I6) holds. (I6) could be falsified only by statements $6.p$ (which establishes $p@\{7\}$ and updates $p.output$) and $5.q$ (which updates *Final*), where q is any process. Statement $6.p$ establishes $p.output = Final$. For statement $5.q$, we have the following.

$\text{I6} \ \wedge \ Enabled(5.q)$

$\Rightarrow \ (\text{I6} \ \wedge \ p@\{7\} \ \wedge \ Enabled(5.q)) \ \vee \ (\neg p@\{7\})$
, predicate calculus.

$\Rightarrow \ (p.output = Final \ \wedge \ p@\{7\} \ \wedge \ Enabled(5.q)) \ \vee \ (\neg p@\{7\})$
, by the definition of (I6).

$\Rightarrow \ (p.output = Final \ \wedge \ Final \neq \bot \ \wedge \ Enabled(5.q)) \ \vee \ (\neg p@\{7\})$
, by (I1).

$\Rightarrow \ (p.output = Final \ \wedge \ Final = Propose \ \wedge \ Enabled(5.q)) \ \vee \ (\neg p@\{7\})$
, by (I5) and the definition of $Enabled(5.q)$.

$\Rightarrow \ (p.output = Final \ \wedge \ Final = q.temp \ \wedge \ Enabled(5.q)) \ \vee \ (\neg p@\{7\})$
, by (I4) and the definition of $Enabled(5.q)$.

$\Rightarrow$ $wp(5.q,\ \text{I6})$, by the definition of (I6) and the monotonicity of wp. $\Box$

The correctness of the program in Fig. 1 can now be easily argued. By (I1), (I2), and (I6) we have

- $p@\{7\} \ \Rightarrow \ (\exists r :: p.output = r.input)$.

In addition, by (I6), we have,

- $p@\{7\} \ \wedge \ q@\{7\} \ \Rightarrow \ p.output = q.output$.

To see this, consider the following derivation.

$$
\begin{aligned}
& p@\{7\} \ \wedge \ q@\{7\} \\
\Rightarrow \ & p.output = Final \ \wedge \ q.output = Final && \text{, by (I6).} \\
\Rightarrow \ & p.output = q.output && \text{, predicate calculus.}
\end{aligned}
$$

(As an aside, note that (I6) implies that the value of *Final* cannot be altered once some process has terminated. In particular, if $p@\{7\}$ holds, then the value of $p.output$ cannot be altered, as it is local to process p. By (I6), $p.output = Final$ holds as well, which implies that the value of *Final* also cannot be altered.)

From the results of this section, we have the following theorem and corollary.

Theorem 2 *In a priority-scheduled uniprocessor system, the consensus problem can be solved in constant time for any number of processes using only atomic read and write operations.*

Corollary 3 *Atomic read and write operations are universal in priority-scheduled uniprocessor systems.*

4 Concluding Remarks

We have shown how to establish invariance properties of programs in priority-scheduled systems. As a test case, we presented and proved correct a simple wait-free consensus algorithm for priority-scheduled uniprocessor systems. This algorithm is one of a series of results by us and colleagues on the application of wait-free and lock-free shared objects in priority- and quantum-scheduled uniprocessor and multiprocessor systems. (Lock-free objects are similar to wait-free objects, except that they do not ensure starvation freedom.) Other papers on this topic include [1–7,12]. This research has resulted in a number of new algorithmic techniques for implementing wait-free and lock-free objects in which characteristics of priority and quantum schedulers are exploited. These techniques can be applied to obtain object implementations that are more efficient than is possible in an asynchronous system.

References

[1] J. Anderson, R. Jain, and K. Jeffay. Efficient object sharing in quantum-based real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 346–355. December 1998.

[2] J. Anderson, R. Jain, and D. Ott. Wait-free synchronization in quantum-based multiprogrammed systems. In *Proceedings of the 12th International Symposium on Distributed Computing*, pages 34–48. Springer Verlag, September 1998.

[3] J. Anderson, R. Jain, and S. Ramamurthy. Wait-free object-sharing schemes for real-time uniprocessors and multiprocessors. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pages 111–122. December 1997.

[4] J. Anderson and M. Moir. Wait-free synchronization in multiprogrammed systems: Integrating priority-based and quantum-based scheduling. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 123–132. May 1999.

[5] J. Anderson and S. Ramamurthy. A framework for implementing objects and scheduling tasks in lock-free real-time systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 92–105. December 1996.

[6] J. Anderson, S. Ramamurthy, and R. Jain. Implementing wait-free objects in priority-based systems. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–238. August 1997.

[7] J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free objects. *ACM Transactions on Computer Systems*, 15(6):388–395, May 1997.

[8] E. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[9] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[10] M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[11] R. Rajkumar. *Synchronization In Real-Time Systems – A Priority Inheritance Approach.* Kluwer Academic Publishers, Boston, 1991.

[12] S. Ramamurthy, M. Moir, and J. Anderson. Real-time object sharing with minimal support. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 233–242. May 1996.

[13] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time system synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.