Wait-Free Synchronization in Multiprogrammed Systems: Integrating Priority-Based and Quantum-Based Scheduling

(Extended Abstract)

James H. Anderson*
Department of Computer Science
University of North Carolina at Chapel Hill

Mark Moir†
Department of Computer Science
University of Pittsburgh

Abstract

We consider wait-free synchronization in multiprogrammed uniprocessor and multiprocessor systems in which "hybrid" schedulers are employed that use both priority information and a scheduling quantum in making scheduling decisions. The main contribution of this paper is to show that, in any hybrid-scheduled system, any object with consensus number $C \geq P$ in Herlihy's wait-free hierarchy is universal for any number of processes executing on P processors, provided the scheduling quantum is of a certain size. We also show that if a C-consensus object must be "hard-wired" to the processors that access it, then our characterization of the required quantum is asymptotically tight. If $C = P$ or if $C \geq 2P$, then this characterization is asymptotically tight regardless of whether objects must be "hard-wired".

1 Introduction

This paper is concerned with wait-free synchronization in multiprogrammed systems. A *multiprogrammed system* consists of a set a processes and a set of processors. Each process is assigned to a distinct processor. Associated with each processor is a *scheduler*, which determines when each process on that processor is allowed to execute. In related previous work, Ramamurthy, Moir, and Anderson considered wait-free synchronization in multiprogrammed systems in which processes on the same processor are scheduled by priority [5]. In a priority-scheduled system, each scheduler always selects for execution the highest-priority process that is available for execution on its processor. Ramamurthy et al. showed that, for priority-based systems, any object with consensus number $C \geq P$ in Herlihy's wait-free hierarchy [4] is universal for any number of processes executing on P processors, i.e., universality is a function of the number of *processors* in the system, not the number of *processes*.

In subsequent work, Anderson, Jain, and Ott considered multiprogrammed systems in which quantum-based scheduling is used [1]. Under quantum-based scheduling, each processor is allocated to its assigned processes in discrete time units called *quanta*. When a processor is allocated to some process, that process is guaranteed to execute without preemption for Q time units, where Q is the length of the quantum, or until it terminates, whichever comes first. Anderson et al. showed that, for quantum-scheduled systems, any object with consensus number $C \geq P$ in Herlihy's wait-free hierarchy is universal for any number of processes executing on P processors, *provided* the quantum is of a certain size.

In this paper, we present results that extend and unify the earlier work cited above. Specifically, we consider multiprogrammed systems in which "hybrid" schedulers are employed that use both priority information and a scheduling quantum in making scheduling decisions. In a hybrid-scheduled system, the scheduler on each processor always chooses for execution a process of maximal priority on its processor. However, there may be several processes of the same priority on the same processor. Processor time is allocated among such processes using a scheduling quantum. Hybrid-scheduled systems are not merely of theoretical interest. Indeed, a number of commercially-available operating systems schedule processes in this way. Examples include the QNX real-time operating system, SGI's IRIX REACT/Pro, and Wind River's VxWorks (all real-time operating systems; see http://www.qnx.com/, http://www.sgi.com/real-time/, and http://www.wrs.com, respectively). In addition, al-

*Work supported by NSF grants CCR 9510156 and CCR 9732916, and an Alfred P. Sloan Research Fellowship.
†Work supported in part by an NSF CAREER Award, grant number CCR 9702767.

C	universal if:
P	$Q \geq c(P+1)T$
$P+1$	$Q \geq cPT$
$\cdots$	$\cdots$
n	$Q \geq c(2P+1-n)T$
$\cdots$	$\cdots$
$2P-2$	$Q \geq c3T$
$2P-1$	$Q \geq c2T$
$2P$	$Q \geq c2T$
$\cdots$	$\cdots$
m	$Q \geq c2T$
$\cdots$	$\cdots$
∞	$Q \geq 0$

Table 1: Conditions for which an object with consensus number C is universal in a P-processor hybrid-scheduled system.

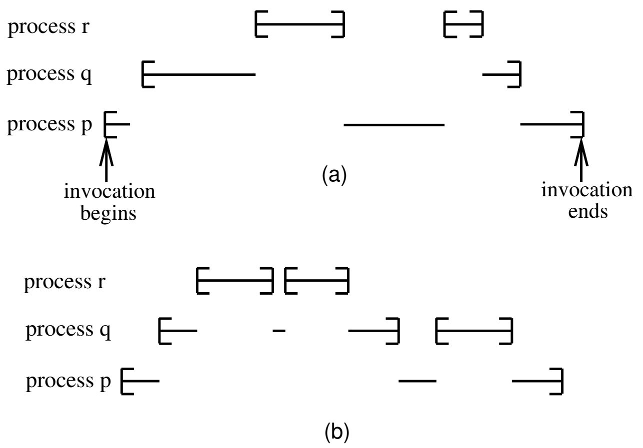

Figure 1: Accesses to a common object by three processes running on the same processor (a) in a quantum-based system and (b) in a priority-based system. Object invocations are shown between the brackets [and], with time running from left to right. In (b), process r (p) has highest (lowest) priority.

gorithms for hybrid-scheduled systems have the interesting property of being correct in systems that are *either* purely priority-based *or* purely quantum-based.

We show in this paper that results established previously for quantum-based systems can be extended to also apply to hybrid-scheduled systems. In particular, we show that, in any hybrid-scheduled system, any object with consensus number $C \geq P$ in Herlihy's wait-free hierarchy is universal for any number of processes executing on P processors, provided the scheduling quantum is of a certain size. We also show that if a C-consensus object must be "hard-wired" to the processors that access it, then our characterization of the required quantum is asymptotically tight. A *hard-wired* object can be accessed only via "ports" that are statically allocated to processors. If $C = P$ or if $C \geq 2P$, then our characterization of the required quantum is asymptotically tight regardless of whether objects must be hard-wired.

All of the algorithms we present to establish these results are of polynomial space and time complexity. In contrast, the main multiprocessor algorithm given previously by Ramamurthy et al. for priority-based systems (a subclass of the hybrid systems we consider) requires exponential space and time. Our results are summarized in Table 1. This table gives conditions under which an object with consensus number C is universal in a P-processor hybrid-scheduled system. In this table, T is a constant denoting the maximum time required to perform any atomic operation, Q is the length of the scheduling quantum, and c is a constant that follows from the algorithms we present. Obviously, if $C < P$, then universal algorithms are impossible [4]. If $P \leq C \leq 2P$, then an object with consensus number C is universal if Q is a value proportional to $(2P+1-C)T$. If $C = \infty$, then Q (obviously) can be any value [4].

To see the difference between a priority-based and a quantum-based system, consider Fig. 1. In this figure, process interleavings are shown that might arise on a processor in a quantum-based system (inset (a)) and in a priority-based system (inset (b)). The interleavings are depicted in a manner that abstracts away from the activities of the processes outside of object invocations — a less abstract look at the quantum-based interleaving in inset (a) is shown in Fig. 2. In a priority-based system, if a process p is preempted during an object invocation by another process q that invokes the same object, then p "knows" that q's invocation must be completed by the time p resumes execution. This is because q has higher priority and will not relinquish the processor until it completes. Thus, operations of higher-priority processes "automatically" appear to be atomic to lower priority processes executing on the same processor. This is the fundamental insight behind the results of Ramamurthy et al. [5]. In contrast, in a quantum-based system, if a process is ever preempted while accessing some object, then there are no guarantees that the process preempting it will complete any pending object invocation before relinquishing the processor. However, if a process can ever detect that it has "crossed" a quantum boundary, then it can be sure that the next few instructions it executes will be performed without preemption. Many of the quantum-based algorithms presented by Anderson et al. in [1] employ such a detection mechanism. In the algorithms presented in this paper, mechanisms are incorporated to allow processes to deal with *both* priority-based *and* quantum-based preemptions.

The remainder of this paper is organized as follows. In Sec. 2, we present definitions and notation used in the remainder of the paper. Then, in Sec. 3, we present a wait-free, constant-time uniprocessor implementation of a consensus object that uses only reads and writes. This implementation is correct as long as the quantum is larger than a specified constant. This result shows that reads and writes are universal in a hybrid-scheduled

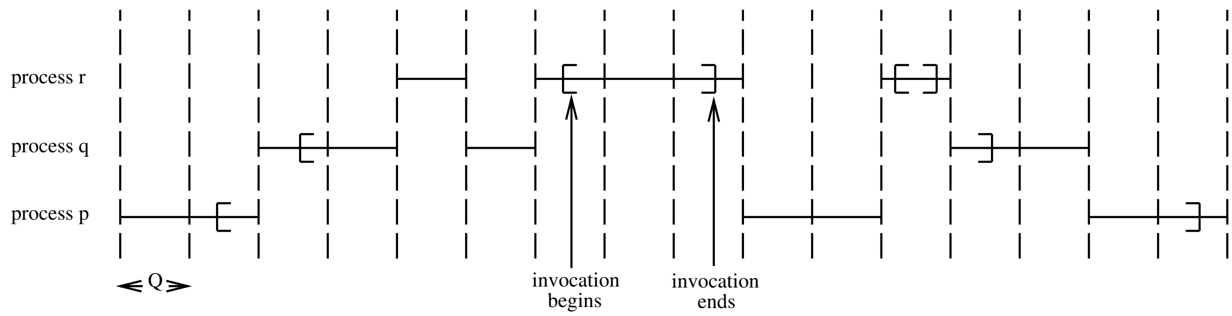

Figure 2: A closer look at the interleaving in Fig. 1(a).

uniprocessor. In Sec. 4, we present our results for hybrid-scheduled multiprocessor systems. We begin by presenting a wait-free implementation of a consensus object for any number of processes running on P processors. This implementation employs objects with consensus number C, where $C \geq P$. As C varies, the quantum required for this consensus implementation to work correctly is as given in Table 1. In the last part of Sec. 4, we show that if C-consensus objects must be hard-wired, then our characterization of the required quantum is asymptotically tight. We also show that for $C = P$, our characterization of the required quantum is asymptotically tight regardless of whether objects must be hard-wired. Note that asymptotic tightness trivially follows for $C \geq 2P$, because in this case, the required quantum is a constant. We end the paper with concluding remarks in Sec. 5. Due to space limitations, only proof sketches are provided for some of our results. Complete proofs can be found in the full paper [2].

2 Preliminaries

In this section, we describe the schedulers considered in this paper, and then formally define our execution model. We consider pure priority- and quantum-based schedulers, and also hybrid schedulers. Hybrid schedulers generalize both priority- and quantum-based schedulers, so we present formal definitions for hybrid systems only.

In a priority-scheduled system, each process on a processor is assumed to have a distinct priority, and each scheduler always selects for execution the highest-priority process that is available for execution on its processor. For simplicity, we will assume throughout most of this paper that all process priorities are statically defined. However, most of the algorithms we present can be easily modified to work correctly in systems that allow priorities to changed dynamically, as long as a process's priority cannot change *during* an object invocation. We consider the issue of supporting dynamic priorities in more detail in Sec. 5.

Associated with any quantum-scheduled system is a *scheduling quantum* (or *quantum* for short), which is a nonnegative integer value. In an actual quantum-based system, the quantum would be given in time units. In this paper, we find it convenient to more abstractly view a quantum as specifying a statement count. This allows us to avoid having to incorporate time explicitly into our model. In a pure quantum-scheduled system, when a processor is allocated to some process, that process is guaranteed to execute without preemption for at least Q atomic statements, where Q is the value of the quantum, or until its current object invocation terminates.

In a hybrid-scheduled system, there may be several processes of the same priority on the same processor. A hybrid scheduler will always choose a process of maximal priority on its processor for execution. If there are several such processes, then processor time is allocated among them using a scheduling quantum. In defining the behavior of a hybrid scheduler, there a variety of different ways in which one can specify the manner in which higher-priority processes may preempt lower-priority ones. In this paper, we mostly limit our attention to hybrid schedulers satisfying the following two axioms.

Axiom 1: (*Priority-Based Scheduling*) If a higher-priority process p becomes ready for execution while a lower-priority process q is executing on the same processor, then p immediately preempts q, i.e., q may be preempted before finishing a full quantum. □

Axiom 2: (*Quantum-Based Scheduling*) Preemptions by higher-priority processes do not affect the quantum-based scheduling at each priority level. Specifically, each process p is guaranteed to execute at least Q statements between preemptions by processes of equal priority, even if p is preempted by higher-priority processes. □

One might be tempted to modify Axiom 1 so that higher-priority processes may preempt lower-priority ones only at quantum boundaries. However, as we shall see, the lower bounds on quantum size that we establish do not depend on Axiom 1. Furthermore, any algorithm that works correctly assuming Axiom 1 is also correct if preemptions occur only at quantum boundaries.

One might also be tempted to eliminate Axiom 2. Indeed, one of the overriding goals of this paper is to consider how various nuances of hybrid schedulers can be exploited to achieve wait-free synchronization, and a hybrid scheduler that violates Axiom 2 is perfectly reasonable. It turns out, however, that hybrid schedulers satisfying Axiom 1 but violating Axiom 2 can be quickly dispensed with, because in this case it can be easily shown that Herlihy's hierarchy remains intact.[1] To see this informally, consider a collection of processes on one processor, with two priority levels. Suppose the lower-priority processes all access a common consensus object, and the higher-priority processes are engaged in some unrelated computation that does not involve that object. A scheduler violating Axiom 2 can allow higher-priority processes to preempt lower-priority ones at arbitrary points, creating a run that is essentially asynchronous from the standpoint of the lower-priority processes. This implies that consensus numbers in such a system have the same interpretation as in an asynchronous system.

From the explanation in the previous paragraph, it may seem that we enforce Axiom 2 only because systems that satisfy this axiom are more "interesting". However, we shall see that, with Axiom 2 in place, it is possible to achieve wait-free synchronization using primitives that are weaker than those given by Herlihy's hierarchy. Thus, if one is interested in supporting wait-free synchronization in a system, then our results suggest that designing that system in accordance with Axiom 2 might be a good idea. As a final defense of these axioms, we note that any algorithm designed in accordance with Axioms 1 and 2 will work correctly in *either* a pure priority-scheduled system *or* a pure quantum-scheduled systems. In some sense, such algorithms are resilient to the specific type of scheduler used in a multiprogrammed system.

Throughout most of this paper, we do not constrain the manner in which the scheduler on each processor allocates quanta to processes of the same priority. Indeed, a scheduler on some processor may choose to *never* allocate a quantum to some ready process. Such behavior corresponds to a "halting failure" in an asynchronous system. In Sec. 5, we briefly consider schedulers that are constrained to allocate quanta "fairly".

Given the above discussion, we can now more carefully define our model of program execution for hybrid-scheduled systems. Our programming notation should be self explanatory; as an example of this notation, see Fig. 3. In this and subsequent figures, each numbered statement is assumed to be atomic. When considering a given object implementation, we consider only statement executions that arise when processes perform operations on the given object, i.e., we abstract away from the other activities of these processes outside of object accesses. For objects that are "long lived", i.e., may be invoked repeatedly by a process, we abstractly view each process as alternating between *thinking* and *ready* phases. While *thinking*, a process p has no enabled atomic statements. A transition by p from *thinking* to *ready* is caused by the scheduler on p's processor. When such a transition occurs, an operation of the implemented object is selected nondeterministically and p's program counter is updated to point to the first statement of that operation. p transits from *ready* to *thinking* when it completes an object invocation.

We define a program's semantics by a set of histories. A *history* of a program is a sequence $t_0 \xrightarrow{s_0} t_1 \xrightarrow{s_1} \cdots$, where t_0 is an initial state and $t_i \xrightarrow{s_i} t_{i+1}$ denotes that state t_{i+1} is reached from state t_i via the execution of statement s_i; unless stated otherwise, a history is assumed to be a maximal such sequence. Consider the history $t_0 \xrightarrow{s_0} t_1 \xrightarrow{s_1} \cdots t_i \xrightarrow{s_i} t_{i+1} \cdots t_j \xrightarrow{s_j} t_{j+1} \cdots$, where s_i and s_j are successive statement executions by some process p. We say that *p is preempted before s_j* in this history iff some other process on p's processor executes a statement between states t_{i+1} and t_j. A history $h = t_0 \xrightarrow{s_0} t_1 \xrightarrow{s_1} \cdots$ is *well-formed* iff it satisfies the following condition: for any statement execution s_j in h by any process p,

- no higher-priority process on p's processor has an enabled statement at state t_j;
- if p is preempted before s_j, then no other process on p's processor with the same priority as p executes a statement after state t_{j+1} until either (i) p executes at least Q statements or (ii) p's object invocation that includes s_j terminates.

We henceforth assume all histories are well-formed.

Notational Conventions: The number of processes and the number of processors in the system are denoted N and P, respectively. Processors are labeled from 1 to P. We assume that the priority levels on each processor range from 1 to V, with V being the highest priority. We let M denote the maximum number of processes on any processor. Q denotes the value of the quantum, and C will be used to refer to a given object's consensus number (see Sec. 1). Unless stated otherwise, p, q, and r are assumed to be universally quantified over process identifiers. We use *valtype* to denote an arbitrary type. □

3 Uniprocessor Systems

In this section, we present a constant-time implementation of a consensus object for hybrid-scheduled systems. This implementation uses only reads and writes and is correct if the quantum exceeds a certain constant value.

Uniprocessor consensus can be solved using the algorithm in Fig. 3. This algorithm was originally proposed for quantum-scheduled systems (as noted in [1], the algorithm is due to Moir and Ramamurthy, but they did not publish it). We show here that it is also correct

[1]Note that a system that allows preemptions only at quantum boundaries satisfies Axiom 2 by definition.

in hybrid-scheduled systems. The algorithm is correct provided Q is large enough to ensure that each process can be quantum-preempted at most once during the **for** loop. By replacing the **for** loop by straight-line code, it can be seen that $Q = 8$ suffices. The algorithm employs three shared variables, $P[1]$, $P[2]$, and $P[3]$. The idea behind the algorithm is to attempt to copy a value from $P[1]$ to $P[2]$, and then from $P[2]$ to $P[3]$. Each process returns the value in $P[3]$. Correctness follows from the following lemma.

Lemma 1: *In the consensus algorithm in Fig. 3, each process returns the same value.* □

Proof: Suppose towards a contradiction that two processes return different values $v1$ and $v2$. Then, some process $s1$ must write $v1$ to $P[3]$ and some process $s2$ must write $v2$ to $P[3]$. Fig. 4 depicts what a history leading to such a situation must look like. Specifically, we have the following:

(A) $s1$ reads $P[3] = \perp$ and then
(B) writes $P[3] := v1$ and
(C) $s2$ reads $P[3] = \perp$ and then
(D) writes $P[3] := v2$. This implies that
(E) $s1$ either reads $P[2] = v1$ or writes $P[2] := v1$ before (A) and
(F) $s2$ either reads $P[2] = v2$ or writes $P[2] := v2$ before (C).
(G) We assume, without loss of generality, that $E < F$.
(H) Some process $r1$ writes $P[2] := v1$ at or before (E). (N.B. The dashed line indicates that it could be that $r1 = s1$ and (H)=(E).)
(I) Also, some process $r2$ writes $P[2] := v2$ after (E) and at or before (F). (N.B. The dashed line indicates that it could be that $r2 = s2$ and (I)=(F).)
(J) $r2$ previously reads $P[2] = \perp$ (must be before (H) because $P[i] \neq \perp$ is stable).
(K) Similarly, $r1$ previously reads $P[2] = \perp$.
(L) $r1$ reads $P[1] = v1$ or writes $P[1] := v1$ before (K).
(M) $r2$ reads $P[1] = v2$ or writes $P[1] := v2$ before (J).
(N) In the diagram, we have assumed $(L) < (M)$. The rest of the proof is symmetric for the other case.
(O) Some process p writes $P[1] := v1$ at or before (L). (N.B. The dashed line indicates that it could be that $p = r1$ and (O)=(L).)
(P) Some process q writes $P[1] := v2$ after (L) and at or before (M). (N.B. The dashed line indicates that it could be that $q = r2$ and (P)=(M).)
(Q) q reads $P[1] = \perp$ before (P) and before (O) because $P[i] \neq \perp$ is stable.

Observe that process p takes a step ((O)) between two steps of process q ((Q) and (P)). Therefore, $priority(p) \geq priority(q)$. If $priority(p) > priority(q)$, then p completes execution before q resumes, which contradicts $P[2] = \perp$ at (J). Thus, $priority(p) = priority(q)$. Similarly, $priority(r1) = priority(r2)$. (Note that (J) < (H) < (I) and that if $r1$ completed execution before (I), $P[3] \neq \perp$ would hold at (C).)

If $priority(q) > priority(r2)$, then q would run to completion before (M), contradicting $P[2] = \perp$ at (J). Also, if $priority(q) = priority(r2)$, then q would run to completion before (H), contradicting $P[3] = \perp$ at (C). To see this, observe that q has already been preempted once by a process of the same priority (namely p) before (P). Therefore, q runs to completion before any other process of the same priority runs again. However, because process $r1$ executes a statement at (H) and because $priority(r1) = priority(r2)$, this contradicts $P[3] = \perp$ at (C). Therefore, $priority(q) < priority(r2)$, which implies that $priority(q) < priority(r1)$ (recall that $priority(r1) = priority(r2)$). But q executed a statement ((P)) between two statement executions of $r1$ ((L) and (H)). Contradiction. □

```
shared variable P: array[1..3] of valtype ∪ ⊥   initially ⊥

procedure decide(val: valtype) returns valtype
private variable v, w: valtype

v := val;
for i := 1 to 3 do
    w := P[i];
    if w ≠ ⊥ then
        v := w
    else
        P[i] := v
    fi
od;
return P[3]
```

Figure 3: Consensus algorithm for hybrid uniprocessors.

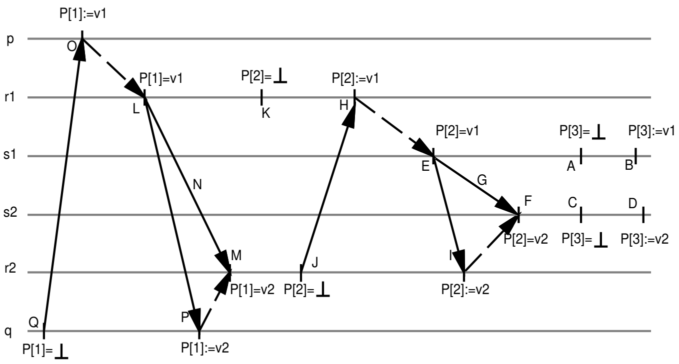

Figure 4: Proof of Lemma 1.

Theorem 1: *In a hybrid-scheduled uniprocessor system with $Q \geq 8$, consensus can be implemented in constant time using only reads and writes.* □

4 Multiprocessor Systems

In this section, we consider multiprocessor systems with hybrid schedulers. In Sec. 4.1, we present a multiprocessor consensus algorithm based on C-consensus primitives that is correct provided the quantum Q is as specified in Table 1. Then, in Sec. 4.2, we show that our characterization of the required quantum is asymptotically tight if $C = P$ or if C-consensus objects must be hard-wired.

4.1 Multiprocessor Consensus

In this subsection, we establish the bounds on Q given in Table 1 for hybrid-scheduled multiprocessor systems. Specifically, we present a wait-free consensus algorithm for any number of processes executing on P processors, where the processes on each processor are scheduled using a hybrid scheduler. The algorithm uses C-consensus objects, where $C \geq P$, and requires the quantum Q to be as specified in the table. For simplicity, we assume in the rest of this section that $C < 2P$, because for larger values of C, the algorithm we give for $C = 2P - 1$ can be applied to obtain the results in Table 1.

Our consensus algorithm is shown in Fig. 5. This algorithm has been obtained by modifying a similar algorithm presented previously in [1] for purely quantum-scheduled systems. The modifications allow multiple priority levels to be supported. In addition to C-consensus objects, the algorithm employs a number of uniprocessor compare-and-swap (`C&S`), fetch-and-increment (`F&I`), and consensus objects. We use "*local*-`C&S`", "*local*-`F&I`", and "*local-consensus*" in Fig. 5 to emphasize that these are uniprocessor objects. Operations on these uniprocessor objects can be implemented in constant time using algorithms for quantum-scheduled systems given in [1].

We begin our description of the algorithm by giving a very brief overview of it; details are provided in the following paragraphs. The algorithm works by having each process participate in a series of "consensus levels". There are L consensus levels, as illustrated in Fig. 6. L is a function of M, P, and C, as described below. Each consensus level consists of a C-consensus object, where $C = P + K$, $0 \leq K < P$. Also associated with each consensus level l is a collection of shared variables $Outval[i, l]$, where $1 \leq i \leq P$. A process on processor i "publishes" the decision value of level l by recording it in $Outval[i, l]$, and by then updating another shared variable, as described below, to indicate that a published value exists at this level (see lines 32-33 in Fig. 5). The processes on each processor attempt to progress through all levels in sequence, using published results from previous levels as inputs to subsequent levels. Each process begins by skipping over any levels for which lower-priority processes executing on the same processor have already published results (lines 5-13). The remaining levels (if any) are accessed in sequence until an overall decision can be reached (lines 14-36). Note that if a process is preempted by a higher-priority process, then by the time it resumes execution, an overall decision must have already been reached. Such a situation is checked for in line 16. If not preempted by a higher-priority process, a process returns as its decision value the output value of the highest-numbered level with a published value.

The requirement that at most C processes can access a C-consensus object is enforced by defining $P+K$ "ports" per consensus level. Processors 1 through K have two ports per object, and processors $K + 1$ through P have one port. A process can access a C-consensus object only by first claiming one of the ports allocated to its processor (note that we are using hard-wired C-consensus objects here). A process claims a port by executing lines 17-26. On each processor, all ports across all consensus levels are numbered in sequence, starting at 1. A process p on processor i claims a port by first incrementing a counter $Port[i, v]$, where v is its priority, and by then invoking a local consensus object associated with that port. The *Port* counters ensure that at most one process of a given priority attempts to acquire a given port. The local consensus invocation ensures that, if several processes with different priorities attempt to acquire a port, then only one succeeds. It is easy to see that a process can fail to "win" such a local consensus invocation only if it is preempted by a higher-priority process. In this case, when the process resumes execution, it will quickly detect that a decision has been reached and terminate. If a process with priority v is *not* preempted by higher-priority processes, then each time it increments $Port[i, v]$, it *atomically* reserves the next available port.

We now describe the manner in which the *Port* variables are incremented. If processor i has one port per C-consensus object, i.e., $i > K$, then $Port[i, v]$ is updated by simply performing a `F&I` operation. If processor i has two ports per object, i.e., $i \leq K$, then the situation is a bit more complicated. In particular, if a process p acquires the first of processor i's ports at some level l and then simply increments $Port[i, v]$, p is then positioned to acquire the second of processor i's ports at level l, which is pointless because a decision has already been reached at that level. To correct this, $Port[i, v]$ is updated in such a case using a `C&S` operation in line 21. Because $Port[i, v]$ is updated only by processes with the same priority, the `F&I` and `C&S` operations used in updating it can be implemented from reads and writes using the constant-time algorithms for purely quantum-scheduled systems presented in [1]. The local consensus object for each port can also be implemented from reads and writes in constant time, using the algorithm given in Sec. 3.

Having explained how ports are acquired, we now explain in more detail how an overall decision value is reached. Each process attempts to participate in each consensus level, in order, skipping over levels for which a decision value has already been published. When a process accesses some level, the input value it uses is either the output of the highest-numbered consensus level for which there is a published value on its processor, or its own input value, if no previously-published value exists (see lines 4, 27, and 28). So that an input value for a level can be determined in constant time, a counter $Lastpub[i, v]$ is maintained on each processor i, for each priority level v. This counter points to the highest-

```
constant L = (K + P)M(P − K + 1) + (P − K)^2 M + 1        /* total number of consensus levels for C = P + K, 0 ≤ K < P */

shared variable
   Lastpub: array[1..P, 1..V] of 0..L  initially 0;          /* Lastpub[i, v] is the highest level on a processor i ... */
                     /* ... for which some process with priority v (or lower) has published a consensus value */
   Outval: array[1..P, 1..L] of valtype ∪ ⊥  initially ⊥;    /* Outval[i, l] is the consensus value for level l on processor i ... */
                     /* ... we assume no input value (and hence consensus value) is ⊥ */
   Port: array[1..P, 1..V] of 1..2L + M  initially 1         /* Port[i, v] is the next available port for processes of priority v ... */
                     /* ... on processor i (each processor has L or 2L ports; the "+M" term is needed because of overshoots) */

private variable                                                                  /* local to process p */
   input, output, lastval: valtype;              /* input/output value for a level, output value of last level */
   level, prevlevel: 0..L + M;                   /* consensus level accessed by p in current (previous) iteration of the while loop */
   numports: 1..2;                               /* number of ports per consensus object on processor pr(p) */
   port, newport, lowerport: 1..2L + M;          /* port numbers */
   publevel, lowerpublevel: 0..L;                /* used to determine last level for which there is published value on processor pr(p) */
   v: 1..V

procedure decide(val: valtype) returns valtype
1:  lastval := Outval[pr(p), L];
    /* return if other processes have already decided */
2:  if lastval ≠ ⊥ then return(lastval) fi;
3:  if pr(p) ≤ K then numports := 2 else numports := 1 fi;
4:  input, prevlevel, level := val, 0, 0;
    /* lower-priority processes may have made some        */
    /* progress, so initialize Port & Lastpub accordingly */
5:  for v := 1 to priority(p) − 1 do
6:     lowerport := Port[pr(p), v];
7:     port := Port[pr(p), priority(p)];
8:     if lowerport > port then
9:        local-C&S(&Port[pr(p), priority(p)], port, lowerport)
       fi;
10:    lowerpublevel := Lastpub[pr(p), v];
11:    publevel := Lastpub[pr(p), priority(p)];
12:    if lowerpublevel > publevel then
13:       local-C&S(&Lastpub[pr(p), priority(p)],
                                      publevel, lowerpublevel)
       fi
    od;
    /* decide (continued) — proceed through levels */
14: while level ≤ L do
15:    lastval := Outval[pr(p), L];
       /* return, if HP processes preempt and decide */
16:    if lastval ≠ ⊥ then return(lastval) fi;
       /* determine port and level */
17:    port := Port[pr(p), priority(p)];
18:    level := ((port − 1) div numports) + 1;
       /* decide (continued) — if level didn't change, make correction */
19:    if prevlevel = level then
20:       newport := port + numports;
21:       if local-C&S(&Port[pr(p), priority(p)], port, newport + 1) then
22:          port := newport
23:       else port := local-F&I(&Port[pr(p), priority(p)])
24:       fi
       else
25:       port := local-F&I(&Port[pr(p), priority(p)])
       fi;
26:    level := ((port − 1) div numports) + 1;
       /* determine input for next level */
27:    publevel := Lastpub[pr(p), priority(p)];
28:    if publevel ≠ 0 then input := Outval[pr(p), publevel] fi;
29:    if level ≤ L then
          /* local consensus ensures at most one process accesses port */
30:       if local-consensus(pr(p), port, p) = p then
             /* invoke the C-consensus object */
31:          output := C-consensus(level, input);
32:          Outval[pr(p), level] := output;   /* publish result */
33:          local-C&S(&Lastpub[pr(p), priority(p)], publevel, level)
          fi
       fi;
34:    prevlevel := level
    od;
35: publevel := Lastpub[pr(p), priority(p)];
36: return(Outval[pr(p), publevel])
```

Figure 5: Multiprocessor consensus algorithm.

numbered level for which a process on processor i with priority v (or lower) has published a value. Due to preemptions, $Lastpub[i, v]$ may need to be incremented to skip over an arbitrary number of levels. It is therefore updated using a C&S operation instead of F&I (see line 33). Like $Port[i, v]$, $Lastpub[i, v]$ is updated only by processes with priority v, so the C&S operations that update $Lastpub[i, v]$ can be implemented from reads and writes in constant time [1]. If a process on processor i is preempted by higher-priority processes, or if a decision is reached before it starts, then it may safely return the output value from level L (lines 2 and 15). Otherwise, it completes execution by returning the output value from level $Lastpub[i, v]$, where v is its priority (lines 35-36).

The main difficulty associated with our algorithm is that of determining an input value to use when accessing a given level. When a process p attempts to determine an input value, there may be a number of previous levels that are inaccessible to p (because all ports on p's processor have been claimed) yet no decision value has been published. This can happen for a previous level l only if the process(es) on p's processor that acquired that processor's port(s) at level l were preempted before publishing an output value (in particular, such a process must have been preempted while executing within lines 21-33). We call such a situation an *access failure*, and say that the preempted process(es) at level l *cause an access failure at level* l. We call an access failure a *same-priority* (*different-priority*) *access failure* if the processes involved (i.e., p and the process(es) preempted at level

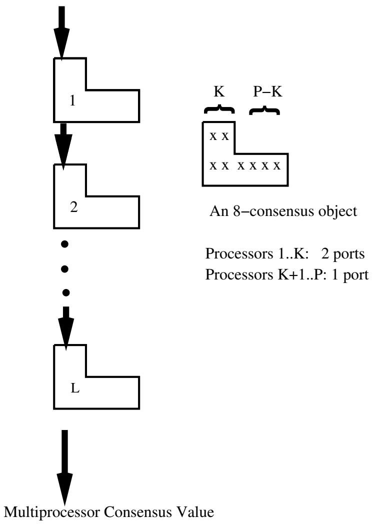

Figure 6: Consensus levels of the algorithm in Fig. 5.

l) have the same priority (different priorities).

Obviously, there is a correlation between the number of access failures that can occur on a processor and the number of preemptions that can occur on that processor. The number of preemptions that can occur in turn depends on the number of processes per processor and the size of the scheduling quantum Q. We show below that with a suitable choice of Q, the number of levels for which an access failure occurs on each processor is limited to a fraction of all the levels. Using a pigeon-hole argument, it is possible to show that if the number of levels L is selected appropriately, then in any history, there exists some level for which no process on any processor experiences an access failure. We call such a level a *deciding level*. A simple inductive argument shows that, if l is a deciding level, then the output value of level l is used by *every* process on *every* processor when accessing any level $k > l$, even if access failures occur when accessing levels numbered higher than l (i.e., levels below l in Fig. 6). Thus, the final decision value returned by all processes is the same.

We now state some lemmas about the number of access failures that may occur in a history. For each lemma, we assume that the given history is fixed. Let AF denote the number of levels for which there is an access failure in this history. Further, let AF_{same} (respectively, AF_{diff}) denote the number of levels for which there is a same-priority (respectively, different-priority) access failure in the history. It follows that $AF \leq AF_{same} + AF_{diff}$, because a preemption could potentially result in *both* a same-priority and different-priority access failure.

Each process can cause a different-priority access failure at at most one level. In particular, if a process p causes a different-priority access failure at level l, then it was preempted by a higher-priority process on its processor while accessing level l. By the time p resumes execution, higher-priority processes have already reached a decision, so p terminates without accessing any more levels. This argument yields the following bound for AF_{diff}.

Lemma 2: *If there are at most M processes on any processor, then $AF_{diff} \leq PM$.* □

In the next lemma, we concentrate on bounding AF_{same}. In the statement of this lemma, we consider a process to *access* a level iff it successfully acquires a port for that level. A proof of this lemma is given in [2].

Lemma 3: *Suppose that $C = P+K$, where $0 \leq K < P$, and that Q is large enough to ensure that each process can be preempted at most once by processes of equal priority while accessing any $P - K + 1$ consensus levels in succession (these levels don't have to be consecutive). If there are L levels, and at most M processes on any processor, then $AF_{same} \leq KM + (P - K)(L + M(P - K))/(P - K + 1)$. Furthermore, if $L > (K+P)M(P-K+1)+(P-K)^2M$, then a deciding level exists.* □

It follows from Lemma 3 that with L as defined in Fig. 5, the algorithm is correct. It is easy to see that each consensus level is accessed in constant time (recall that the uniprocessor `C&S`, `F&I`, and consensus algorithms used at each level take constant time). Thus, letting the constant c denote the worst-case number of statement executions per level, we have the following theorem.

Theorem 2: *In a P-processor system in which the processes on each processor are scheduled using a hybrid scheduler, consensus can be implemented in a wait-free manner in polynomial space and time for any number of processes using read/write registers and C-consensus objects if $C \geq P$ and $Q \geq \max(2c,\ c(2P + 1 - C))$.* □

If we were to incorporate time within our model, then we could easily incorporate the T term given in Table 1 in the bound on Q given in Theorem 2.

4.2 Lower-Bound Results

In this subsection, we show that if $C = P$, then our characterization of the quantum required for universality is is asymptotically tight. As explained below, with only a few modifications, the proof strategy given here can be used to show that, if C-consensus objects must be hard-wired, as in the algorithm of the previous subsection, then all of the bounds on Q given in Table 1 are asymptotically tight. Due to space limitations, we only sketch the main ideas of the $C = P$ proof here. The complete proof can be found in [2].

We restrict attention to systems with schedulers that are purely quantum-based, i.e., we assume there is only one priority level per processor. Clearly, lower bounds for such a system will apply to a system with multiple priority levels as well. In the remainder of this subsection, we let A denote a consensus algorithm for an arbitrary number of (deterministic) processes executing on a P-processor system with a single priority level per processor such that all underlying objects are either read/write registers or P-consensus objects. For simplicity, we assume that any invocation of any of these underlying objects takes one atomic statement. We also assume that, in any history, if a P-consensus object is invoked more than P times, then all invocations after the P-th return the value $\perp$.[2] Our objective is to show that A cannot be correct in a system for which $Q = P$. Assume, to the contrary, that A *is* correct in such a system. We show that this assumption leads to a contradiction.

Our proof is based on a valency argument [3, 4], and we assume that the reader is familiar with the general structure of such an argument. In particular, we assume that the reader knows what it means for a state of A to be *bi-valent*, *uni-valent*, or *x-valent*, where x is the input value of some process [3, 4]. We derive a contradiction by showing that there exists a history of A consisting of an infinite sequence of bi-valent states, contradicting the fact that A is wait-free. The proof is similar to other valency arguments presented elsewhere [3, 4], with two important exceptions. First, we must carefully keep track of when a given process may be preempted. Second, we define the valency of a state with respect to the set of currently-running processes at that state. In other words, the valency of a state t is defined by considering *only* histories starting from t that include statement executions of the P processes that are currently running at t.

We assume that there are exactly two processes on each processor. We denote the processes assigned to processor i as p_1^i and p_2^i. The initial set of running processes is defined to be $\{p_1^1, p_1^2, \ldots, p_1^P\}$. Because the set $\{p_1^1, p_1^2, \ldots, p_1^P\}$ consists of P processes and P is the size of the quantum, we can "stagger" the execution of these processes with respect to quantum boundaries so that, of these processes, one has executed at least $P-1$ statements since last being preemptable, another has executed at least $P-2$ statements since last being preemptable, and so on. We call a bi-valent state at which this "staggered execution" property holds *strong-bi-valent*.

In the full proof, we inductively show that an infinite sequence of bi-valent states exists by proving that from any strong-bi-valent state we can reach another strong-bi-valent state. With $\{p_1^1, p_1^2, \ldots, p_1^P\}$ as the set of currently-running processes, as we move from state to state, we can fail to extend this inductive argument only if there exists a reachable strong-bi-valent state t, at which each of $p_1^1, p_1^2, \ldots, p_1^P$ is enabled to invoked a common consensus object O, such that if any one of $p_1^1, p_1^2, \ldots, p_1^P$ takes a step from t, the resulting state is

[2] We could instead require that such invocations return random values, or that such invocations are in fact not allowed. The key requirement here is that all invocations after the P-th return no "useful" information.

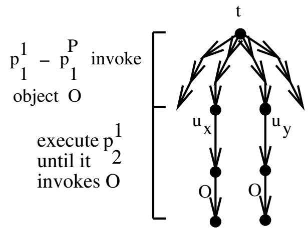

Figure 7: Two histories ending in x-valent and y-valent states. Process p_2^P cannot distinguish between the two histories by invoking object O, so it chooses the same decision value in both histories, which is a contradiction.

uni-valent (we emphasize that such a state is considered to be uni-valent with respect to the set of running processes $\{p_1^1, p_1^2, \ldots, p_1^P\}$). Furthermore, among the states reachable from t by having each of $p_1^1, p_1^2, \ldots, p_1^P$ take a step, there is an x-valent state u_x and a y-valent state u_y, where $x \neq y$ (see Fig. 7).

Because t is strong-bi-valent, of the processes in $\{p_1^1, p_2^1, \ldots, p_1^P\}$, one has executed at least $P-1$ statements since last being preemptable, another has executed at least $P-2$ statements since last being preemptable, and so on. Thus, at states u_x and u_y, one of these processes has executed at least P statements since last being preemptable, another has executed at least $P-1$ statements since last being preemptable, and so on. Moreover, the process that has executed at least P statements since last being preemptable can be preempted once again. Without loss of generality, assume this preemptable process is p_1^1.

Suppose we change the currently-running set of processes at state u_x by replacing process p_1^1 by p_2^1. If u_x is bi-valent for this new set of running processes, then we have reached yet another strong-bi-valent state (with a different set of running processes than before) and the induction continues. If, on the other hand, u_x is uni-valent for this new set of running processes, then it must be x-valent, because we know that if we run, say, process p_1^2 in isolation from u_x, it returns x as its decision value. In this case, if we run p_2^1 in isolation from u_x, it too must eventually choose x as its decision value.

All of the reasoning in the previous paragraph can be applied to state u_y as well, i.e., if we define $\{p_2^1, p_1^2, p_1^3, \ldots, p_1^P\}$ to be the set of currently-running processes at u_y, then state u_y is either bi-valent or y-valent for this set of processes. In the former case, the induction continues, and in the latter case, p_2^1 must choose y as its decision value if it is run in isolation from u_y.

To finish the proof, note that if we run p_2^1 in isolation from both u_x and from u_y, then it must return the same decision value in both histories. This is because object O has consensus number P and has already been invoked P times. Thus, p_2^1 cannot distinguish between the two histories. We conclude that, with $\{p_2^1, p_1^2, p_1^3, \ldots, p_1^1\}$ defined as the set of currently-running processes, at least one of u_x and u_y is bi-valent, and therefore the inductive argument can be continued, i.e., A is not wait-free.

If C-consensus objects must be hard-wired, then a virtually identical proof shows that all of the bounds on Q given in Table 1 are asymptotically tight. In this case, as we move from state to state, we maintain the invariant that $P-Q+1$ processes exist that have executed at least $Q-1$ statements since last being preemptable, and of the remaining processes, one has executed at least $Q-2$ statements since last being preemptable, another has executed at least $Q-3$ statements since last being preemptable, and so on. As before, we can fail to inductively extend the history being constructed for the current set of running processes only if a situation as depicted in Fig. 7 is eventually reached. In this case, of the $P-Q+1$ processes that have executed at least $Q-1$ statements before t since last being preemptable, at least one, say p_1^1, does not have a port associated with object O. For exactly the same reasons as in the $C=P$ proof, if we change the current set of running processes by replacing p_1^1 by p_2^1, then the induction can continue. The results of this subsection give us the following theorem.

Theorem 3: *In a P-processor quantum- or hybrid-scheduled system, there is no wait-free consensus algorithm for an arbitrary number of processes using read/write registers and C-consensus objects* **(i)** *if $C=P$ and $Q \leq P$, and* **(ii)** *if the C-consensus objects are hard-wired and $C \geq P$ and $Q \leq \max(1,\ 2P-C)$.* □

```
shared variable
    Output: valtype ∪ ⊥   initially ⊥

procedure decide(val: valtype) returns valtype
private variable
    output: valtype

if local-consensus(pr(p), priority(p), p) ≠ p then
    while Output = ⊥ do od;
    return(Output)
fi;
output := global-PB-consensus(val);
Output := output;
return(output)
```

Figure 8: Multiprocessor consensus with fair scheduling.

5 Concluding Remarks

We have heretofore assumed that all task priorities are statically defined. However, the multiprocessor consensus algorithm given in Sec. 4 can be easily extended to support dynamic priorities. We can obtain the identifiers required by the algorithm by using a uniprocessor renaming algorithm that assigns the same name to same-priority processes. Our uniprocessor consensus algorithm is correct in a dynamic-priority system as stated, so we know that reads and writes are universal in a hybrid-scheduled uniprocessor with dynamic priorities. As a result, it is straightforward to show that the required renaming object can be implemented.

Our definition of a hybrid scheduler assumes that such a scheduler can "unfairly" allocate quanta to processes of the same priority. If we assume that quanta are "fairly" allocated, then P-consensus primitives can be used to implement consensus for any number of processes on P processors, using a quantum of constant size. To see this, consider the multiprocessor consensus algorithm given in Fig. 8. In this algorithm, we first "elect" one process per priority level on each processor by means of invoking a local uniprocessor consensus object for that level/processor. If a process loses the election for its priority level, then it waits until the other processes choose a decision value. Because the quantum-based scheduling on each processor is fair, each process waits for only a finite time. The election winners all invoke a multiprocessor priority-based consensus algorithm to determine a decision value. One may feel that the algorithm in Fig. 8 violates the spirit of wait-freedom. However, if we define wait-freedom to mean that each process completes an operation in a finite number of its own steps, then this algorithm is indeed wait-free.

Acknowledgements: Srikanth Ramamurthy helped in developing the algorithm in Fig. 3. Eli Gafni initially suggested to us the possibility of designing wait-free algorithms that work correctly in both priority- and quantum-based systems.

References

[1] 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*, pp. 34–48. Springer Verlag, Sept. 1998.

[2] J. Anderson and M. Moir. Wait-free synchronization in multiprogrammed systems: integrating priority-based and quantum-based scheduling (expanded version of this paper). Available at `http://www.cs.unc.edu/~anderson/papers.html`.

[3] M. Fischer, N. Lynch, and M. Patterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

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

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